Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS

Karthikeyan Bhargavan, Benjamin Beurdouche, Prasad Naldurg

Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS

Karthikeyan Bhargavan Benjamin Beurdouche Prasad Naldurg
Inria Paris

***Abstract*—Group conversations are supported by most modern messaging applications, but the security guarantees they offer are significantly weaker than those for two-party protocols like Signal. The problem is that mechanisms that are efficient for two parties do not scale well to large dynamic groups where members may be regularly added and removed. Further, group messaging introduces subtle new security requirements that require new solutions. The IETF Messaging Layer Security (MLS) working group is standardizing a new asynchronous group messaging protocol that aims to achieve strong guarantees like forward secrecy and post-compromise security for large dynamic groups.**

In this paper, we define a formal framework for group messaging in the F⋆ language and use it to compare the security and performance of several candidate MLS protocols up to draft 7. We present a succinct, executable, formal specification and symbolic security proof for TreeKEM$_B$, the group key establishment protocol in MLS draft 7. Our analysis finds new attacks and we propose verified fixes, which are now being incorporated into MLS. Ours is the first mechanically checked proof for MLS, and our analysis technique is of independent interest, since it accounts for groups of unbounded size, stateful recursive data structures, and fine-grained compromise.

I. Introduction

With the rise in popularity of instant messaging applications like WhatsApp, Skype, and Telegram for both personal and business interactions, the security and privacy of messaging conversations has become a pressing concern. Many of these applications have adopted sophisticated cryptographic protocols that provide end-to-end guarantees against powerful attackers. For example, WhatsApp and Skype use the Signal protocol [1], while Telegram relies on MTProto [2]. For a full survey of messaging protocols and their properties, see [3].

At an abstract level, a messaging protocol has the same goals as a secure-channel protocol like Transport Layer Security (TLS). However, messaging scenarios have several distinguishing characteristics, leading to different protocol designs. First, messages are *asynchronous*: a user needs to be able to send messages to her interlocutors even if they are offline. In practice, this means that messaging applications must rely on servers to store and forward messages, making these servers attractive attack targets. Furthermore, conversations are *long-lived*: unlike TLS connections, which typically last for seconds, messaging conversations may continue for months and have thousands of sensitive messages. As a result, there is a significant risk that one of the endpoints may be broken into or confiscated during the lifetime of a conversation.

If an attacker gains control over one of the endpoints, it will be able to read any messages stored on the device, but it should not be able to read older ciphertexts (beyond some time interval) that it may have obtained earlier; this guarantee is usually called *forward secrecy* (FS). Furthermore, if the attacker only temporarily compromises an endpoint, the protocol should be able to lock out the attacker and allow the victim to rejoin and *heal* the conversation, a guarantee specific to secure messaging called *post compromise security* (PCS).

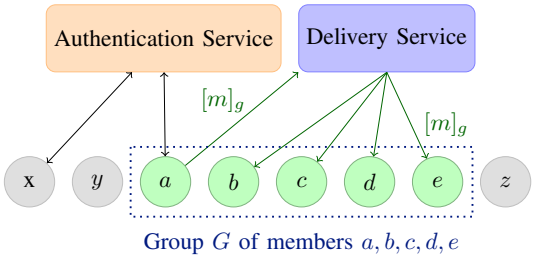

Fig. 1. Group Messaging Architecture: Devices obtain credentials from a trusted authentication service to form messaging groups within which they can exchange end-to-end encrypted messages via an untrusted delivery service.

There are many messaging protocols that achieve these security goals for two-party conversations. For example, Signal establishes initial encryption keys between two devices using an asynchronous key exchange protocol called X3DH [4]. Thereafter, the protocol aggressively updates the encryption keys as often as possible, using a mechanism called the Double Ratchet [5] that provides both FS and PCS.

However, most messaging applications also support *group conversations* whose security guarantees are less well understood. Messaging groups can have hundreds of members who join and leave over time. Even messages between two users may turn into small group conversations, if users are allowed to register *multiple devices*. Such multi-party scenarios have new security requirements that need novel protocol designs, but have received relatively little attention in the literature [3].

Secure Group Messaging Requirements. A typical group messaging architecture is depicted in Figure 1. Device a sends a confidential message m (via some delivery service) to a messaging group g that has five members $\{a, b, c, d, e\}$. We assume that these members can authenticate each other using credentials issued by some trusted authentication service. The attacker controls the network, the delivery service, and can dynamically compromise any device.

In this scenario, the main group message security requirements are as follows: The message m can only be read by the members of the group $\{a, b, c, d, e\}$ and nobody else (*Message Confidentiality*). When a group member b receives m, it knows which member a sent the message, and if a is uncompromised, the attacker cannot have tampered with the message (*Message & Sender Authenticity*). Further, each member of the group knows (and agrees on) who else is in the group; we do not

allow ghost users [6] (*Group Agreement*).

Importantly, we expect these security requirements to continue to hold even when groups change over time. When a new member is added to the group, it can send and receive new group messages, but it should not be able to read older messages (*Add Security*). Similarly, when a group member is removed, it should be immediately locked out, unable to read or send new group messages (*Remove Security*). A member who suspects that its keys may be compromised should be able to update its keys at any time, thereby protecting newer messages from past compromise (PCS), and older messages from future compromise (FS) (*Update Security*).

Many of these properties are specific to group messaging. In two-party conversations, there is only one peer that needs to be authenticated, and users can easily abandon conversations if they detect an attack. Groups are harder to set up, and so even if a member is actively compromised, i.e. a *malicious insider*, we expect to be able to remove the member and continue the group conversation, without tearing down the group. Although prior work on two-party [3] and group messaging [7] have considered post-compromise security against passive compromise, our formalization of Remove Security against malicious insiders is new, to the best of our knowledge.

IETF Messaging Layer Security (MLS). Despite the extensive literature on group key exchange protocols, there are not many protocols that achieve the functional or security requirements we have laid out for group messaging. Most academic group protocols (e.g. [8]) are not designed for the asynchronous setting. WhatsApp deploys a protocol called Sender Keys, which provides FS, but not PCS, has limited support for dynamic groups, and uses up to $O(N^2)$ pairwise Signal channels to establish a fully connected group.

In 2017, the Internet Engineering Task Force (IETF) established a working group to design a secure group messaging protocol that scales well to large dynamic groups. The first version of MLS (draft 0) was based on a protocol called Asynchronous Ratcheting Trees (ART) [7], which incorporates ideas from tree-based group key management and embeds them within an asynchronous messaging protocol that provides FS and PCS. The protocol requires $O(N)$ Diffie-Hellman (DH) operations to set up a group of size N and only $O(log(N))$ DH operations to send and process group updates.

In draft 2, ART was replaced by TreeKEM [9] that is also based on a tree data structure, but reduces the processing time for group changes and updates to $O(1)$ public key operations at recipients. TreeKEM relies on a generic public-key encryption (PKE) construction, that can be instantiated using DH or many other (e.g. post-quantum) encryption schemes.

Both ART and TreeKEM suffer from an attack called the *Double Join*, where a malicious member can insinuate itself at multiple locations in the group, making it hard to remove, hence breaking Remove Security. To protect against such active attacks, MLS draft 7 specifies a protocol called *TreeKEM with Blanking* (TreeKEM_B) trading performance for security. However, neither TreeKEM nor TreeKEM_B have been formally analyzed, which makes it hard to precisely evaluate their security guarantees, and compare them to other proposals.

In this paper, we present the first detailed formal specifications and mechanized security proofs for multiple protocols considered for adoption in MLS (up to draft 7). We explain their design choices and formally relate the Double Join attack to the inability to remove a group member who is a malicious insider. Our work has already had a major impact on the MLS protocol. The design of both TreeKEM and TreeKEM_B were influenced by our preliminary analyses. We also found new attacks on TreeKEM_B, and we have proposed verified countermeasures that are in the process of being incorporated into the next draft of MLS.

Succinct, Executable, Formal Specifications for MLS. A protocol standard like MLS serves both as a high-level textual description of the protocol, as well as a low-level implementation guide with enough details to guarantee interoperability between implementations. To be useful and credible, a formal specification of the protocol must be succinct, readable, and contain all details of the protocol. Furthermore, the specification should be machine-checkable, so that we can quickly find modeling mistakes, and so that the specification can serve as a basis for mechanized security proofs.

We present a formal specification for MLS written in the F* programming language and verification framework [10]. Our specification is compact, detailed, and executable, and serves as both a formal companion and a reference implementation of the standard, against which other implementations can be tested. We use F* as a verifier to build a machine-checked symbolic proof that our MLS specification meets our desired security requirements, also formalized in F*.

A Symbolic Security Analysis of MLS draft 7. Formal security proofs of protocols are generally classified in two categories: (1) *computational* proofs with precise complexity-theoretic probabilistic assumptions about the underlying cryptographic primitives; (2) *symbolic* or Dolev-Yao analyses that rely on algebraic abstractions of primitives. Symbolic proofs are easier to mechanize and can provide valuable semi-automated feedback on low-level details of frequently evolving standards like MLS. Computational proofs, usually require significant manual effort and are useful in their own right to analyze the cryptographic core of the protocol. Both methods can be used to provide complementary benefits, as demonstrated in recent analyses of TLS 1.3 [11], [12].

In this work, we focus on developing a symbolic security proof for a formal specification of MLS draft 7 in F*, following the type-based verification methodology of [13], [14]. The first challenge in building a security proof for MLS is that we need to account for groups of arbitrary size, where each member maintains a stateful recursive tree data structure. This kind of protocol is typically out of the reach of automated analysis tools like ProVerif [15] or Tamarin [16], but well-suited for F*. A second challenge is to be able to model fine-grained compromise to prove properties like FS and PCS, which have not been formalized before in F*.

We address these challenges and present the first mechanized security proofs for various MLS candidates, accounting for arbitrarily large groups with malicious insiders, and an unbounded sequence of messages and group changes. Our

analysis helped us uncover weaknesses and attacks, and to verify our proposed fixes. We believe that our framework offers a strong basis for evaluating future changes to MLS.

II. Messaging Layer Security Requirements

The MLS Architecture document [17] lays out an architecture and a list of requirements for group messaging protocols. In this section, we present a formal framework for group messaging in the F⋆ programming language [10] and use it to precisely specify these functional and security requirements.

A. An MLS Protocol API

Figure 2 presents an F⋆ interface that each messaging protocol must implement to meet the functional requirements of MLS. The interface consists of a sequence of types and functions for group management and encrypted messaging.

Principals and Group Members. A participant in a messaging protocol is called a *principal* (written $a, b, \ldots$), and each principal can obtain credentials in its name from some trusted authentication service. For example, this credential may be an X.509 certificate issued by some public key infrastructure. We assume that each credential is associated with a signature key (sk_a) that the principal can use to authenticate its messages.

Within a group, each member principal is identified by a member_info record, which consists of a credential and a public encryption key. Each member is expected to regularly update this encryption key, and the member_info includes a version number identifying the current key (ek_a^v).

The datatype member_secrets represents the secrets corresponding to a member_info, notably the signature key (sk_a) and current decryption key (dk_a^v).

Group States. Each member of a messaging group stores and maintains a local copy of the public group state, represented by the type group_state. Every messaging protocol implements group_state with its own data structure, but provides functions to read the group identifier, the maximum group size, and the current membership, defined as a member_array: an array where each index is either unoccupied (None) or contains a member_info (Some mi). Since the membership of the group can change, the group state also includes an *epoch* field that indicates the current version of the group as a whole.

Any principal can locally create a new group state by calling the function create and providing a fresh group identifier, a maximum size, an initial membership, and some protocol-specific key material (entropy) that can be used to generate a shared group secret. The optional return type indicates that this function may fail (returning None), say if one of the member_info entries in init has an invalid credential or a non-zero version or if the entropy is insufficient, but if it succeeds, it returns a group state g with the desired parameters.

Once a group state has been created, the creator will typically send it to all other members within a Create message, and each recipient will validate the state and store it locally if it is willing to join the proposed group.

Group Operations. A group member can modify its local group state by constructing and applying an operation.

```
(* Public Information about a Group Member *)
type member_info = {
  cred: credential;
  version: nat;
  current_enc_key: enc_key}
(* Secrets belonging to a Group Member *)
val member_secrets: datatype
(* Group State Data Structure *)
val group_state: datatype
val group_id: group_state → nat
val max_size: group_state → nat
val epoch: group_state → nat
type index (g:group_state) = i:nat{i < max_size g}
type member_array (sz:nat) =
      a:array (option member_info){length a = sz}
val membership: g:group_state → member_array (max_size g)
(* Create a new Group State *)
val create: gid:nat → sz:pos → init:member_array sz
         → entropy:bytes → option group_state
(* Group Operation Data Structure *)
val operation: datatype
(* Apply an Operation to a Group *)
val apply: group_state → operation → option group_state
(* Create an Operation *)
val modify: g:group_state → actor:index g
         → i:index g → mi':option member_info
         → entropy:bytes → option operation
(* Group Secret shared by all Members *)
val group_secret: datatype
(* Calculate Group Secret *)
val calculate_group_secret: g:group_state → i:index g
         → ms:member_secrets → option group_secret
         → option group_secret
```

```
(* Protocol Messages *)
type msg =
      | AppMsg: ctr:nat → m:bytes → msg
      | Create: g:group_state → msg
      | Modify: operation → msg
      | Welcome: g:group_state → i:index g
                → secrets:bytes → msg
      | Goodbye: msg
(* Encrypt Protocol Message *)
val encrypt_msg: g:group_state → gs:group_secret
         → sender:index g → ms:member_secrets → m:msg
         → entropy:bytes → (bytes * group_secret)
(* Decrypt Initial Group State *)
val decrypt_initial: ms:member_secrets
         → c:bytes → option msg
(* Decrypt Protocol Message *)
val decrypt_msg: g:group_state → gs:group_secret
         → receiver:index g → c:bytes
         → option (msg * sender:index g * group_secret)
```

Fig. 2. An F⋆ Interface for MLS Protocols. Each protocol must implement a Group Management and Key Exchange (GMKE) component that establishes a shared group secret (top) and a Message Protection (MP) component that uses the group secret to protect messages (bottom).

Each messaging protocol provides its own data structure for operations, and provides two functions: modify creates an operation, and apply executes the operation to a group state.

Each operation is authored by a principal, called the *actor*, who wishes to modify the membership array at some index. The actor may *add* a new member at an unoccupied index, or *remove* an existing member, or *update* its own member_info record by providing a new credential or encryption key

(ek_a^{v+1}). The function modify creates an operation given a group state, the index of the actor, the index to modify, a (possibly empty) member_info to write into that index, and some fresh key material used to refresh the group secret.

After calling modify to construct an operation, the actor applies it to its local group state and sends the operation to the all members in a Modify message, so that they can apply it to their local states. If the operation adds a new member, and that member has no prior group state, the actor sends it the full new group state in a Welcome message. When removing a member, the actor sends it a GoodBye message.

Group Secrets. Each group state is associated with a shared group_secret. A member at index i can calculate this secret using the public group state g and its own member_secrets record ms, by calling the function calculate_group_secret g i ms.

As long as the group state has been created by applying a valid sequence of operations, the group secret calculated by each member should be the same, and this secret s should be known only to the *current members* of the group. In particular, if a member a has an encryption key with version v (ek_a^v), then even if a's previous decryption key (dk_a^{v-1}) is leaked to the attacker, s should remain secret. Similarly, even if a's next decryption key dk_a^{v+1} is eventually revealed to the attacker, the attacker should not be able to recompute s. This *versioned secrecy* requirement for the group secret is at the heart of the group security guarantees (FS, PCS) we expect from a group messaging protocol that meets our API.

Message Protection. During a typical run of the messaging protocol (see Figure 9) group members send a number of messages to each other. The type msg defines a tagged union of these message types: a msg can either contain a group management message (Create, Modify, Welcome, GoodBye), called a *handshake message*, or an application message (AppMsg).

To send a message m, a group member (sender) uses the group state, group secret, its own index and member secrets, and calls the function encrypt_msg, which signs the message with the sender's signature key and encrypts it for the intended recipients. Encryption may change the group secret (e.g. it may increment a counter or ratchet a key) and so it returns both the ciphertext and the new group secret.

When a member receives its first message for a group (Create or Welcome) it calls decrypt_initial with its member_secrets to process the message. It then extracts the group state from the message and stores it locally.

If the receiver already has a group state g and group secret s, it calls decrypt_msg with g, s, and the receiver's index to decrypt the message (AppMsg, Modify or GoodBye). If it receives a Modify, it applies the received operation to g; if it receives a GoodBye it deletes g and s.

Messaging Applications. Extending the interface in Figure 2 to a full messaging application requires many more details, including session state storage, networking functions, a public key infrastructure, etc. which we do not model here. Instead, we focus on the functional and security requirements for the core messaging protocol and verify whether a given protocol meets these requirements. In particular, we state and prove several functional correctness lemmas, including one that states that calling apply g o on a group state g and operation o always succeeds if o was the result of modify applied to the same group state g.

B. Threat Model and Security Goals

The high-level security goal of a messaging protocol is to prevent an adversary from stealing, tampering with, or forging application messages as they are exchanged within a group. In our model, we consider adversaries who control the network and may also compromise some valid members of a group (malicious insiders).

Network Attacker. An active network attacker can inject, intercept, modify, replay, and redirect messages sent between any two principals. In terms of the messaging architecture, it means that the attacker controls the delivery service, and hence can send any message to any group member, and intercept any (encrypted) message sent to a member.

The attacker may also call any function, including cryptographic algorithms, to construct and break down messages, and in doing so, may be able to discover protocol values that were meant to be kept secret. We assume that it cannot guess cryptographic keys or break the underlying crypto algorithms (with non-negligible probability). Our precise modeling of cryptographic assumptions is detailed in Section V.

Compromised Principals. We allow the adversary to dynamically and selectively compromise any version of any group member's secrets. In particular, an adversary can compromise the signature key sk_a corresponding to a member's credential, or it may compromise a specific decryption key dk_a^v, without necessarily compromising dk_a^{v-1} or dk_a^{v+1}.

When stating our security goals, we use the predicate auth_compromised mi to refer to the loss of the signature key corresponding to the credential mi.cred, and the predicate dec_compromised mi to refer to the loss of the decryption key corresponding to mi.current_enc_key. We use compromised to refer to the disjunction of the two cases.

Security Goals. We can now relate our main security goals in terms of the MLS protocol interface. Section V describes how we encode these goals in F* and how we can prove that they hold for a messaging protocol.

Message Confidentiality (Msg-Conf) If a member *a* sends a confidential application message m in a group state g, then the message m remains confidential from the adversary, unless one of the members of g is compromised.

Message Authenticity (Msg-Auth) If a member *b* receives an application message m from a sender *a* over a group g, then *a* is a member of g, and if *a* is not auth_compromised, then the message m was indeed sent by *a* over the same group state g.

Group Agreement (Grp-Agr) If a member *b* processes a Create, Welcome, or Modify message from a sender a, resulting in a group state g, and if *a* is not auth_compromised, then *a* sent the message when its local group state was also g.

Add Security (Add-FS) If a member *a* is added to a group state g with group secret s, resulting in a new group

state g' with secret s', then the old group secret s remains confidential, unless one of the members of g is compromised, even if a was compromised before or after the Add.

Remove Security (Rem-PCS) If a member a is removed from a group state, resulting in a group state g' with group secret s', then s' is confidential from the adversary, unless one of the members in g' is compromised, even if a was malicious (actively compromised) before the Remove.

Update Security (Upd-FS,Upd-PCS) Suppose a member a updates its encryption key from version v to $v+1$ in a group state g with group secret s, resulting in a group state g' with secret s'. Then s (resp. s') is confidential from the adversary, unless one of the members in g (resp. g') is compromised. In particular, s (resp. s') remains secret even if dk_a^{v+1} (resp. dk_a^v) is compromised.

Many of the goals presented here are straightforward. The notion of Upd-PCS is similar to the notions of PCS that have been studied before in two-party and group protocols. It allows a member to *heal* a group after its own old keys have been (passively) compromised. Rem-PCS, however, allows the group to heal itself against malicious insiders, i.e. actively compromised members, and is studied here for the first time.

Limitations. We note that our confidentiality guarantees are conditioned on the passive or active compromise of some of the members of a given group. As group sizes increase, it is fair to assume that some members will be inevitably compromised, and the FS and PCS properties basically capture the best possible security guarantees in this context. To recover from compromise, it is sufficient for members to regularly update their keys, and for actively compromised members to be identified and removed. The exact nature and frequency of updates is hard to predict, and MLS leaves the details of update frequency and user (de-)authorization to the application, and so it does not appear in our model.

There are also many other desirable properties of messaging systems that we do not consider here. None of the above guarantees prevent an active network attacker from *partitioning* the group by only letting some group members communicate with each other. We can only guarantee that members in the same partition have consistent group states. Similarly, we do not explicitly state a privacy goal that would protect the group membership from being known to the authentication or delivery service, a problem we leave for future work.

III. Messaging Layer Security Candidates

In this section, we evaluate a series of group messaging protocols that have been considered by the MLS working group, and we informally compare their designs, performance, and security. By doing so, we elaborate the design decisions that led to TreeKEM_B, the protocol used by MLS draft 7.

Table I lists the protocols and summarizes their main differences. Two of the protocols (Sender Keys, ART) are well known, but the others are detailed here for the first time. We have formally modeled and analyzed all protocols (except Sender Keys) in F*, and we present our full specification and analysis of TreeKEM_B in Sections IV and V.

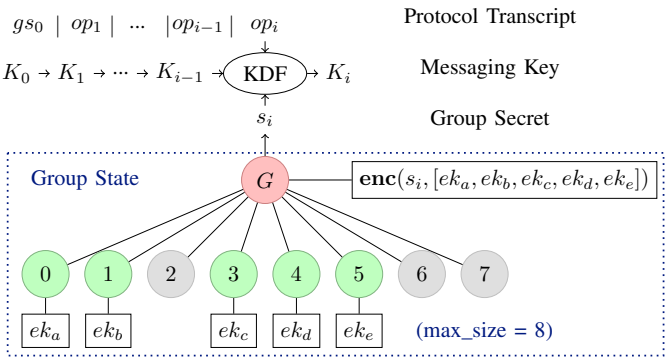

Fig. 3. Chained mKEM: a group state with five members a, b, c, d, e after i operations. The group secret s_i is generated by one of the members and encrypted to the public encryption keys of all current members. The current messaging key K_i is derived from s_i and the previous messaging key K_{i-1}, with the protocol transcript mixed in as additional context.

A. Signal Sender Keys

The Sender Keys protocol [18] is used in Signal and WhatsApp to setup group conversations between principals who already have pairwise Signal channels between them. When a wants to send a message to the group $\{a, b, c, d, e\}$, it generates a fresh *chain key* ck_a and fresh signature key-pair (sk_a, vk_a) and sends (ck_a, vk_a) over Signal to b, c, d, and e, who store these keys locally. The chain key ck_a is used to derive a sequence of symmetric encryption keys $(k_a^0, k_a^1, \ldots)$ that a can then use to encrypt messages to the group. Each encryption key is used only once and then deleted, enabling a message-level forward secrecy (Msg-Conf) guarantee. Each encrypted message is signed with sk_a, providing message and sender authentication (Msg-Auth).

If another member b wishes to send a message to the group, it must also generate and send fresh keys (ck_b, vk_b) to all other members. Hence, to set up a group with N active participants, Sender Keys requires $O(N^2)$ Signal messages to be sent, each of which costs at least one Diffie-Hellman operation. This cost can become prohibitive for large groups.

Sender Keys does not have an explicit notion of groups and hence does not provide Grp-Agr. It also does not provide efficient mechanisms for Update or Remove; to remove a member, a new group needs to be created from scratch. These shortcomings make the protocol unsuitable for MLS.

B. Chained mKEM

Chained mKEM aims to be the simplest protocol that achieves the functional and security requirements of MLS. Although it is not very efficient for large groups, it provides strong baseline security guarantees that we will compare against more efficient protocols described later in this section.

The group_state data structure used in Chained mKEM is depicted in Figure 3. It consists of an explicit membership array, containing optional member_info records for each member. In each epoch i, the group ciphertext containing the group secret s_i is encrypted under the current encryption key of each member (ek_a^v). Given a group state, any member can use its current decryption key (dk_a^v) to decrypt the group secret.

The current group secret s_i is also used to derive a message encryption key K_i that is used to protect group messages

TABLE I
A COMPARISON OF THE COMPUTATIONAL COST AND SECURITY GUARANTEES OF CANDIDATE GROUP MESSAGING PROTOCOLS

Protocol	Create		Add			Remove		Update		Group Agreement	Update PPCS	Remove PACS
	Send	Recv	Send	Recv	New	Send	Recv	Send	Recv			
Sender Keys [18]	N^2	N	1	1	N	-	-	-	-	No	No	No
Chained mKEM$^+$	N	1	1	1	1	N	1	N	1	Yes	Yes	Yes
2-KEM Trees$^+$	N	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	Yes	Yes	No
ART [7]	N	$log(N)$	$log(N)$	$log(N)$	$log(N)$	-	-	$log(N)$	$log(N)$	Yes	Yes	No
TreeKEM$^+$	N	$log(N)$	$log(N)$	1	1	$log(N)$	1	$log(N)$	1	Yes	Yes	No
TreeKEM$_B{}^+$	N	1	1	1	1	$log(N)..N$	1	$log(N)..N$	1	Yes	Yes	No*
TreeKEM$_{B+S}{}^+$	N	1	1	1	N	$log(N)..N$	1	$log(N)..N$	1	Yes	Yes	Yes

Computational Cost (public-key operations): The cost of sending and receiving each group operation (in a group with N active members) is listed in terms of the number of expected public-key operations, including Diffie-Hellman computations, public-key encryptions, and signatures. In all protocols, exchanging a group message requires 1 signature and 1 symmetric encryption. Each member stores the full group state ($O(N)$), and the size of each operation is proportional to the sender's computation cost for that operation.
Group Security Guarantees: All protocols provide mechanisms for message (forward) secrecy, integrity, sender authentication, and Add-FS. We distinguish their security based on whether they provide group agreement (Grp-Agr), Update PPCS, and Remove PACS (i.e. whether they prevent double-join attacks.)
(+): Described in this paper
(*): Section VI describes a double join attack on new members in TreeKEM$_B$ if the member who adds them is malicious.

exchanged within that epoch. Messaging keys are chained together in a key derivation sequence, along with the protocol transcript, to ensure that the current key K_i combines all the group secrets contributed to the group so far.

The core cryptographic construction we need for this protocol is a public key encryption scheme, where the sender generates a fresh key and encapsulates it to a (potentially large) list of recipients. Such a scheme is called a multi-Key Encapsulation Mechanism or mKEM in the literature, following Smart [19]. Since we apply a sequence of mKEMs and chain their result, we call this protocol *Chained mKEM*.

Security. The security of Chained mKEM relies on the invariant that the current group secret s_i is known only to the current members of the group. Whenever the membership changes, even if it is due to an update of a member's key, a new group secret is generated and delivered (only) to the new members, hence maintaining the invariant. The invariant is sufficient to obtain all the secrecy guarantees of MLS. Furthermore, each message is signed along with a hash of the full protocol transcript, providing Msg-Auth and Grp-Agr.

Performance. The cost of sending a group operation in Chained mKEM is 1 mKEM encapsulation, which is usually proportional to N public key encryptions. However, the receiver only needs to perform 1 public key decryption. Add can be implemented more efficiently: the sender encrypts the new group secret s_i using the old group messaging key K_{i-1} (for the old members) and just the new member's encryption key, reducing the sender's cost to 1 public key encryption.

C. A Generic Tree-Based Group Messaging Protocol

In large groups, the $O(N)$ cost of Update in Chained mKEM becomes too expensive, encouraging lower update frequencies, and hence weaker security guarantees. Asynchronous Ratcheting Trees (ART) [7] shows how to reduce the cost of sending group updates from N to $log(N)$ public key operations, by relying on a Diffie-Hellman (DH) construction inspired by tree-based group key agreement (see e.g. [20]). We generalize this idea, so that it is not specific to DH, and then instantiate it with multiple tree-based messaging protocol designs (including ART and TreeKEM$_B$).

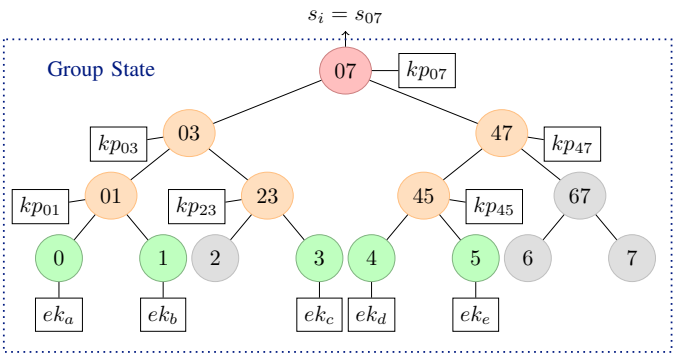

Fig. 4. A Tree of Subgroups: a data structure used in tree-based protocols like 2-KEM Trees, in ART, and TreeKEM. The group has 5 members a, b, c, d, e organized into 6 subgroups 01, 23, 45, 03, 47 and 07; 07 denotes the full group. Each subgroup is associated with a subgroup secret (e.g. s_{01}) and has a key package (e.g. kp_{01}) with the subgroup secret encrypted for its children.

Tree Data Structure. The group state data structure (depicted in Figure 4) now contains a tree of subgroups: members occupy leaves of the tree, and each internal node represents a subgroup consisting of the leaves under it. We consider only *binary leveled trees*, where each internal node has two children, and every leaf is at the same distance from the root. We note that many of the protocols considered here, including TreeKEM$_B$, would also work with other kinds of trees. In F* syntax, a tree with lev levels is defined as follows:

```
type tree (lev:nat) =
 | Leaf : last_actor:credential{lev=0} →
          mi:option member_info → tree lev
 | Node: actor:credential{lev>0} → kp:option key_package →
          left:tree (lev − 1) → right:tree (lev − 1) → tree lev
```

A leaf (e.g. 1) is a tree with 0 levels. It notes the identity of the last actor who modified it, and if the leaf is occupied, it holds a member_info record with a current *leaf encryption key*, written ek_0 or ek_a or ek_a^v, for more specificity.

Each internal node in the tree that has some members under it (e.g. 03 at level 2) is assigned a *subgroup secret* (s_{03}) by the last actor to modify that node. This secret is used to derive a public-key encryption keypair for the subgroup (ek_{03}, dk_{03}). The node holds a key_package (kp_{03}) that includes the encryption key (ek_{03}) and a ciphertext that *encapsulates* (encrypts)

the subgroup secret (s_{03}) under the encryption keys of the child subgroups (ek_{01}, ek_{23}). If an internal node has only one child (e.g. 47), it does not need a key package, it can reuse the subgroup secret and key package of its child ($kp_{47} = kp_{45}$).

The tree data structure is itself public, but each member of the group (e.g. a) can use its current leaf decryption key (dk_0) to compute the secrets for all the subgroups it belongs to: it first *decapsulates* (decrypts) the subgroup secret (s_{01}) for its parent node (01) from the parent's key package (kp_{01}) using its leaf decryption key (dk_0), it then uses the parent secret to derive the corresponding decryption key (dk_{01}) and continues up the tree, decrypting parent secrets until it has the root secret (s_{07}). In practice, each member caches all its subgroup secrets and only recomputes them when they change.

The subgroup secret of the root (s_{07}) is the group secret for the full group (s_i). Like in Chained mKEM, this secret is used to derive a chain of message encryption keys ($K_0, K_1, \ldots$).

Node Encapsulation. The core cryptographic construction used in the generic tree-based protocol here is a *node encapsulation mechanism* that group members can use to construct and process key packages. The mechanism provides two functions:

```
val node_encap: s_c:secret → ek_s:enc_key → dir:direction
                → entropy:bytes → (s_p:secret * kp_p:key_package)
val node_decap: s_c:secret → dir:direction → kp_p:key_package
                → option (s_p:secret)
```

To create a subgroup secret (s_p) and key package (kp_p) for a parent node p, a sender who knows one of the two child subgroup secrets (s_c) can call node_encap with the encryption key of the sibling (ek_s) and some fresh key material (entropy) (dir indicates whether c is the left or right child). Conversely, a receiver can call node_decap with one of the child secrets to decapsulate the parent subgroup secret from a key package.

As we will see, the main difference between different tree-based protocols is in their implementation of these functions.

Group Operations. To create the initial group state, the creator constructs the full tree bottom up, level by level. It first populates the leaves with the initial members. For each parent of a leaf, it calls node encapsulation to generate a subgroup secret and key package, goes up a level, and so on, until it reaches the root. Hence, creating a full tree requires $N = 2^{\mathrm{lev}}$ calls to node encapsulation.

However, each change to the group membership only affects a single *path* from a leaf to the root, which consists of $\mathrm{lev} = log(N)$ nodes in the tree, as defined by the F* datatype:

```
type path (lev:nat) =
 | PLeaf: mi:option member_info{lev=0} → path lev
 | PNode: kp:option key_package{lev>0} →
          next:path (lev−1) → path lev
```

Each path is a sequence of PNodes ending with a PLeaf. To add, remove, or update a member_info at a leaf, a sender first creates a PLeaf with the desired change and repeatedly calls node_encap to create subgroup secrets and key packages for all nodes going up to the root. The resulting path is *applied* as a patch to a local tree, and is also sent (as part of an operation) to other members, who apply it to their own trees. All members then recalculate any subgroup secrets that have changed.

Performance. Creating a tree costs a sender N calls to node_encap, while modifying it costs $log(N)$ calls. At recipients, the main work is to recalculate subgroup secrets, which requires $log(N)$ calls to node_decap. So the cost of each tree-based protocol directly depends on the cost of these functions.

Subgroup Secrecy Invariant. The confidentiality guarantees of all our tree-based protocols rely on a secrecy invariant: each subgroup secret (e.g. s_{03}) can be known only to the current members at the leaves of the subtree. If none of these members is compromised, the secret (s_{03}) cannot be learned by the adversary. Informally, this invariant holds because the subgroup secret is only encapsulated to the encryption keys of its children (ek_{01}, ek_{23}), whose decryption keys are in turn derived from the child subgroup secrets (s_{01}, s_{23}). By applying this reasoning inductively all the way down to the leaves, we obtain the desired secrecy invariant for each node.

From the secrecy invariant on the root secret, we obtain a messaging key K_i that is only known to the current (versions of the current) group members. Using this key to protect group messages is enough to guarantee all the confidentiality goals of MLS (Msg-Conf, Add-FS, Rem-PCS, Upd-FS, Upd-PCS). By requiring the sender's signature on each message, we also obtain the authentication guarantees (Msg-Auth, Grp-Agr).

However, as we show in Section III-E, the secrecy invariant does not always hold if there are malicious insiders, resulting in an attack. The $\mathrm{TreeKEM}_{\mathrm{B}}$ protocol (Section IV) incorporates a fix for this attack, which we analyze in Section V.

D. Three Tree-Based Protocols: 2-KEM Trees, ART, TreeKEM

We now describe three instantiations of the tree-based protocol that define node encapsulation in different ways.

2-KEM Trees. To encapsulate a node, the sender generates a fresh subgroup secret and encrypts it using the public encryption keys of both child subgroups, storing both ciphertexts in the key package. To decapsulate, each member uses one of the child subgroup secrets to decrypt the corresponding ciphertext. This design implements a 2-key version of mKEM, and hence is called 2-KEM Trees. Each call to node_encap costs 2 public-key encryptions and 1 secret-to-public-key derivation; node_decap requires 1 decryption.

ART. Every node's encryption keypair (ek_{03}, dk_{03}) is implemented as a DH keypair $(g^{x_{03}}, x_{03})$. To encapsulate a node (say 03), a sender uses the private key of one child subgroup (x_{01}) and the public key of the sibling ($g^{x_{23}}$) to compute a shared subgroup secret $s_{03} = g^{x_{01}x_{23}}$. It then derives x_{03} (via a key derivation function) from s_{03} and stores $g^{x_{03}}$ in the key package (kp_{03}). To decapsulate the node, a recipient who knows one of the child secrets repeats the DH computation to compute s_{03}. Each call to node_encap requires 1 DH computation and 1 secret-to-public-key derivation; node_decap requires 1 DH computation.

TreeKEM. In both 2-KEM Trees and ART, the subgroup secret is derived in the same way at both the sender and receiver. As a consequence, the cost to the sender and receiver are essentially the same: $log(N)$ public key operations. Although this is much less than Chained mKEM for senders, it represents a significant increase for recipients.

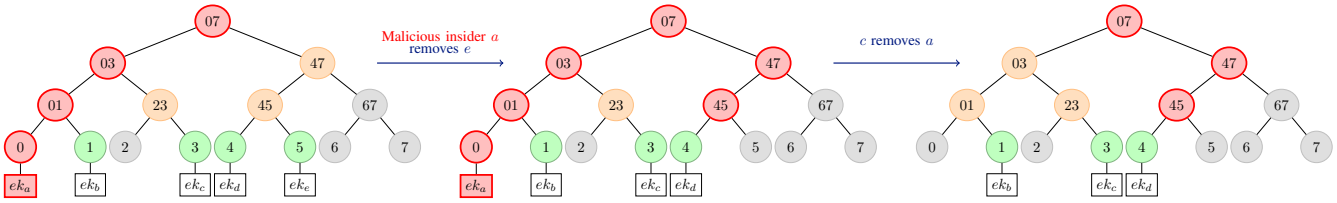

Fig. 5. Double Join Attack on Tree-Based Protocols (2-KEM Trees, ART, and TreeKEM). A malicious (actively compromised) member a (at index 0) removes some member e (at index 5), by generating fresh secrets for the path from 5 to the root. Even if a is subsequently removed (say, by c), it can still compute the new group secret s_{07}, since it knows the secret for 45 and 47. (Red nodes have subgroup secrets that are known to the adversary.)

The key insight of the TreeKEM protocol, first proposed in [9] and adopted in MLS draft 1, is that the cost at the receiver can be reduced to 1 public key operation by using a node encapsulation mechanism that is asymmetric between the two children. A sender who knows one child subgroup secret (say s_{01}) derives the parent subgroup secret (s_{03}) from the child secret (by calling a key derivation function) and then encrypts the derived subgroup secret with the encryption key ek_{23} of the sibling subgroup, as detailed in the F⋆ code below:

```
let node_encap sc eks dir entropy =
    let sp = kdf_derive sc in
    let cs = pke_enc sp eks in
    let dkp = kdf_expand sp "node" in
    let ekp = secret_to_public dkp in
    (sp, mk_key_package ekp dir cs)
```

The node_encap function for TreeKEM returns a key package that contains the node encryption key (ek_p), a direction field (dir) indicating which child subgroup was used to derive the subgroup secret, and a ciphertext for the other child (c_s).

To decapsulate this key package, the recipient looks at dir to determine whether it should derive the parent secret or decrypt it from the ciphertext. In one case, it needs to perform 1 public key decryption, but in the other it only performs 1 key derivation, which is much more efficient.

The impact of this optimization is felt at each group modification, when a recipient (occupying some leaf, say 2) needs to recalculate its subgroup secrets based on a path it has received. The recipient finds its lowest ancestor that lies on the path (say 03) and then decapsulates all the subgroup secrets from this ancestor to the root. In TreeKEM, only the first decapsulation (03) requires a public key decryption, after that, the recipient only needs to use key derivation all the way to the root. Hence, processing a group modification requires only a single public key operation.

E. Malicious Insiders and the Double Join Attack

In MLS, any group member can create a group, and add or remove other members. This means that a member (a) who occupies a leaf (0) in the tree can change the membership of subtree (23) that it is not a member of. In such cases, we say that the member (a) is an *external actor* for the subgroup (23).

In all our tree-based protocols so far, an external actor needs to generate and encapsulate the subgroup secrets for any subgroup it modifies, but we expect it to then *throw away* these subgroup secrets. However, if the external actor is *malicious* (actively compromised), it may hold on to the secrets, so even if it is subsequently removed from the full group, it can still (stealthily) read group messages. Hence, malicious insiders can break the subgroup secrecy invariant.

Consider the scenario in Figure 5. Member a at leaf 0 decides to remove e from leaf 5. To do so, a must generate fresh subgroup secrets for 45, 47, and 07, and encapsulate these secrets in the appropriate nodes. Since a is malicious it keeps the subgroup secrets at these nodes, even though it is not a member of these subgroups. At this point, a can access the group through 2 locations: its official index is 0 but it also has the secrets for leaf 5, a so-called *double join attack*.

In the next step, c at index 3 removes a (perhaps because it detected that a was misbehaving). Now, a is no longer a valid member of the group and does not appear in the group membership. Hence, other group members may think a cannot read messages sent to the group. However, because of the earlier *double join*, a will be able to compute the group secret and read messages. The only way to get rid of a is for d to send an update, or for a new member to be added to leaf 5.

The attack can be extended to extreme cases; a malicious a may remove all other members and then add them all back at the same indexes. From the application's perspective nothing has changed, except that a has double-joined itself to every leaf in the tree, making it hard to remove a from the group.

This double-join attack is a failure of Rem-PCS: none of the tree-based protocols we have considered so far allows a group to remove malicious insider, so the subgroup secrecy invariant holds only if we restrict ourselves to passive compromise. In contrast, note that Chained mKEM is not vulnerable to double join attacks, since it does not allow external actors to set the group secret. Next, we will see how the MLS working group combined ideas from Chained mKEM and TreeKEM to obtain a new, more secure protocol.

IV. TreeKEM$_\mathrm{B}$: Key Establishment in MLS draft 7

TreeKEM with Blanking (or TreeKEM$_\mathrm{B}$) is an extension of TreeKEM that is designed to prevent the Double Join attack. It was first introduced in MLS draft 2 and has evolved in every subsequent draft. We focus on the design in draft 7. Figure 11 in the appendix contains an F⋆ specification for the group management functions in the protocol. In this section, we highlight its main features, referring to the figure for details.

Blank Nodes and the Subgroup Secrecy Invariant. The key idea behind the protocol is that whenever an external actor modifies the subgroup at some node, it does not encapsulate a new subgroup secret for the node; instead, it *blanks* the node by setting the key package to None. Initially, when a creator constructs a new tree, it populates the leaves but blanks all

internal nodes. Subsequently, when a group member adds or removes a leaf at some index i, it modifies the leaf and blanks all the nodes from the leaf to the root, by creating a *blank path* consisting of a leaf and a sequence of blank nodes.

A blank node does not have a subgroup secret and hence trivially satisfies the subgroup secrecy invariant. Conversely, the subgroup secret at each non-blank node must have been generated by one of its members (not an external actor). Hence, blanking prevents the Double Join attack by reestablishing our subgroup secrecy invariant at all nodes.

If a node is blank, it does not have an encryption key, but we can still compute a set of public encryption keys that *covers* the members of the subtree. This is given by the function pub_keys that recursively traverses a tree to extract an array of encryption keys:

```
let rec pub_keys (l:nat) (t:tree l) :
        pks:array enc_key{length pks ≤ pow2 l} =
  match t with
  | Leaf _ None → empty
  | Leaf _ (Some m) → singleton (current_enc_key m)
  | Node _ (Some k) left right → singleton k.node_enc_key
  | Node _ None left right → append (pub_keys (l−1) left)
                                    (pub_keys (l−1) right)
```

For an occupied leaf or a node with a key package, pub_keys returns a singleton array; for an empty subtree, it returns an empty array, and for a blank node, it returns an array of public keys corresponding to the non-blank descendants of the node.

In the worst case, when all the nodes in a tree are blank, pub_keys always returns the set of leaf encryption keys, and the protocol behaves like Chained mKEM. In the best case, when all nodes are non-blank, pub_keys returns a singleton, and the protocol behaves like TreeKEM. So, the cost of encrypting to a subgroup in TreeKEM_B ranges between $log(N)$ and N public key operations.

Unblanking Nodes with Update Paths. The only way to create an non-blank node in TreeKEM_B is for a member to issue an update for its own leaf, by calling the function update_path l t i mi_i s_i. This function computes an *update path* in a tree t with l levels, starting at leaf i. It replaces the member_info at i with mi_i and uses the new leaf secret s_i to encapsulate a sequence of key packages for all internal nodes from i to the root, calling node_encap at each step:

```
let node_encap (s_c:secret) (ek_s:array_enc_key) dir _ =
   let s_p = kdf_derive s_c in
   let c_s = mpke_enc s_p ek_s in
   let dk_p = kdf_expand s_p "node" in
   let ek_p = secret_to_public dk_p in
   (s_p, mk_key_package ek_p dir c_s)
```

The node_encap function for TreeKEM_B generalizes that of TreeKEM by using a list of encryption keys for the sibling, instead of a single key, and calling mpke_enc to encrypt the parent secret for multiple recipients.

Group Operations. At group creation, and after every operation, we need to ensure that the root node is not blank, so that the full group has a valid group secret that can be used to derive messaging keys. After creating the initial (blank) tree, the creator immediately applies an update path from its own leaf (usually at index 0) to the root, hence unblanking a path of nodes including the root. It then uses the resulting tree as the initial group state to other members. Similarly, each group modification operation (Add, Remove, Update) contains two paths: a blank path from the modified leaf to the root, and an update path from the sender's leaf to the root, that restores the root node. Hence, after each operation, the sender and recipients obtain the new group secret and messaging keys corresponding to the new group state.

Calculating the Group Secret. A group member at leaf i in a tree t (with l levels) can use its leaf secret s_i to calculate the root secret for t by calling root_secret l t i s_i, which recursively decapsulates all the subgroup secrets from i to the root, by calling node_decap at each step:

```
let node_decap (s_c:secret) (i:nat) dir kp =
  if dir = kp.from then
    if i ≠0 then None
    else Some (kdf_derive s_c)
  else
    let dk_c = kdf_expand s_c "node" in
    mpke_dec kp.node_ciphertext s_c i
```

To decapsulate a key package kp at some node, a member must either know the child subgroup secret from which the node secret was derived, or it must know one of the decryption keys for the sibling subgroup. The function node_decap takes a secret s_c, an index i, and a direction dir: if dir matches kp.from, node_decap repeats the derivation from node_encap; otherwise, it derives a decryption key from s_c and calls mpke_dec to decapsulate the ciphertext for the ith recipient.

Once a member has calculated the root secret, it then executes a *key schedule* to derive a series of secrets, encryption keys, and nonces for use in protecting protocol messages. We have implemented the key schedule from draft 6 in our model, but in a relatively naïve way in that we only delete messaging keys when the epoch changes, not on a per-message basis. Moreover, this key schedule has undergone significant changes in drafts 7 and 8, in order to achieve stronger message-level forward security guarantees. We plan to faithfully model and verify this new design in future work.

Group States and Transcripts. The group_state data structure in TreeKEM_B consists of the group identifier, tree, an epoch number indicating the number of times the group has been modified, and a transcript hash. Each member in a group conversation has a view of the group's history, called the *protocol transcript*. This transcript begins with a group state, which is either the initial group state in the Create message or a later group state from a Welcome message (if the member was added later). It then continues with a sequence of Modify messages containing group operations.

We say that two transcripts are *consistent* if one of them is a suffix of the other: i.e. the initial state of one corresponds to the initial state or an intermediate state in the other. We show that consistent transcripts result in the same group state, so authenticating the transcript hash is sufficient to guarantee group agreement.

Message Protection. Before sending any protocol message on the network, a sender calls encrypt_msg to protect the message

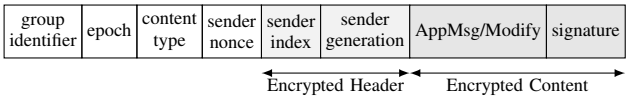

Fig. 6. Encrypted Format for Application Messages and Group Operations

using the most recent key it shares with the recipient. When sending a Create or Welcome message, the sender signs the message (containing the group state), and then encrypts it with the encryption key of the recipient (since the recipient does yet know the group secret). We note that MLS draft 7 actually does not sign these messages, but we consider this an oversight; without these signatures, group agreement immediately fails.

For subsequent messages the sender uses the encrypted message format shown in Figure 6. The message has a header that indicates the group identifier, current epoch number, content type (either AppMsg or Modify), a sender nonce, the sender index, and a counter (generation) for the message.

To protect such messages, encrypt_msg implements a sign-then-encrypt construction. The header and the message contents are first signed (using the sender's signing key); the contents and the signature are then encrypted using an Authenticated Encryption with Associated Data (AEAD) construction. The AEAD encryption uses a sender-specific key and a nonce, both derived as part of the key schedule, and the generation counter, with the header taken as associated data. Finally, the sender's index and generation are also AEAD encrypted (for privacy) using a separate encryption key (also derived from the key schedule) and the sender nonce.

Draft 7 does not include the transcript hash in the signature of AppMsg and Modify, and as we shall see in Section VI, this leads to an attack due to a failure of group agreement.

Stateful Application API. Although not formally part of the MLS standard, each messaging protocol ultimately needs to provide a useful API that can be used by messaging applications. We wrap our protocol code within a stateful API that offers several functions: init creates a new group state, stores the group_state and its group_secrets in the local heap, and returns a group session handle and an encrypted Create message that the application should send to the delivery service; the dual function accept_init takes an encrypted Create or Welcome message, decrypts the group state and its secrets, stores them in the heap, and returns a handle to the application.

Given a group session handle, an application can call get_membership to query the current membership at any time. It can also call add, remove, and update, to modify the group, and send to encrypt an application message, returning the encrypted message to the application. At the recipient, the can application call receive to process an encrypted message. Each of these functions retrieves the local group state and secrets from the heap, performs the protocol operations, and stores the modified group and secrets back in the heap. Importantly, the API does not expose any long-term or short-term protocol secrets to the application, so that the security of the protocol does not depend on the application storing secrets.

A Testable Reference Implementation of MLS. Our full F⋆ implementation of TreeKEM$_B$ including the stateful API, parsing, and serialization functions for operations and group states, constitutes an executable reference implementation of MLS draft 7. We link this code with the HACL* verified cryptographic library [21] to obtain an executable that produces messages that conform with the standard. We have tested that our implementation meets various MLS test vectors, and we intend to test interoperability with other draft 7 implementations developed independently.

```
type prin = string
type sid = p:prin ∗ option (sess:nat ∗ option (ver:nat))
type label = | Public
             | Secret: issuers:list sid → readers:list sid → label
type usage = | AE | PKE | Sig | Nonce | Guid
             | KDF: ext:(label ∗ usage) → exp:usage → usage
```

```
type event =
 | StoreState: p:prin → vv:array nat
               → st:array bytes{length st == length vv} → event
 | Corrupt: id:sid → event
 | GenRand: p:prin → r:bytes → l:label → u:usage → event
 | SendMsg: s:prin → r:prin → m:bytes → event
val trace: append_only_array event
```

Fig. 7. The global execution trace: a *monotonically* growing array of events.

Functional Correctness Lemmas. To gain confidence in our specification, we also prove a series of functional correctness lemmas for our stateful API. In particular, we prove that each function correctly modifies the membership of the group, so that an application can be confident that the membership it sees in a group state correctly reflects the protocol transcript.

V. Formal Security Analysis of TreeKEM$_B$

Each run of an MLS protocol involves messages exchanged between an arbitrary number of principals, each of whom maintains local state that may be independently compromised by an adversary. So, in order to formally prove that our specification of TreeKEM$_B$ meets the security goals of Section II, we first need to define a notion of stateful global executions in F⋆, and then precisely encode both our cryptographic assumptions and the capabilities of the attacker.

A. Modeling Stateful Protocols & Fine-Grained Compromise

We model executions of a protocol in F⋆ as a single stateful global variable called trace that contains an append-only array of *events*, as depicted in Figure 7. As a protocol executes, new events get added to the end of the trace. Hence, the length of the trace provides an abstract notion of global time.

Principals and Session States. Whenever a principal p wishes to store a new state st in its local storage, we model this action by adding an event StoreState p vv st to the global trace. In our model, each principal p has a number of active *sessions* and each session has its own state that evolves over time. So, the stored state st is actually an array of session states (serialized as bytestrings), and is associated with a *version vector* vv that indicates the current version of each session.

In our model of TreeKEM$_B$, each principal has two separate sessions for each group conversation it is a member of. In its so-called *auth session*, it stores its long-term credential and associated signature and initial decryption keys. In its

dec session, it stores its member_info and current decryption key, along with the current group state data and group secrets. The auth session evolves when the member obtains a new credential, whereas the dec session changes with every epoch.

The type sid is used to identify a set of sessions belonging to a principal: it can be used to refer to the principal p as a whole (e.g. id=(p,None)), or can pinpoint a specific version v of a specific session s (e.g. id = (p,Some(s,Some v))).

Corruption Events. Normally, the state st stored by a principal p can only be read by p. However, we allow the attacker to dynamically compromise a specific sid by injecting an event Corrupt id into the trace, which then allows it to read any session state covered by id. For example, if id = (p,Some(s,Some v)), then the attacker only gets to read a specific version v of session s stored by p in the trace.

In our TreeKEM$_B$ model, we define the predicate auth_compromised mi for some member_info mi to mean that the auth session for the credential in mi is corrupted, and dec_compromised mi to meant that the dec session containing this specific version of mi is corrupted.

Corrupt event models dynamic compromise, in that the adversary can choose who to compromise based on the protocol run. Corrupting the current state of a principal models *active* compromise, whereas corrupting some previous version of a session is used to model *passive* compromise. This corruption model allows us to reason about the security of a protocol in various fine-grained compromise scenarios. In particular, we can ask for the forward secrecy (FS) of messages sent before a Corrupt event, or the post-compromise security (PCS) of messages sent after. Although type systems like F* have been used to verify many cryptographic protocols [22], [23], this is the first formalization of FS and PCS in F*.

Generating Fresh Secrets. During the execution of a protocol, a principal p may call a pseudorandom number generator to obtain a fresh random bytestring s for some intended usage u and an expected secrecy level l. This event is recorded as GenRand p s l u in the trace. The usage may range over using the secret as a symmetric encryption key (AE), private decryption key (PKE), a signature key (Sig), a Nonce, a Guid, or a KDF key from which other keys are derived.

The secrecy label l assigned to a GenRand event is a security annotation; it has no impact on the execution of the protocol, but allows to track secret values as they flow through the network, storage, and cryptographic functions. A label indicates whether a value is intended to be Public or Secret. If it is a Secret, it includes a list of issuers and a list of readers. The issuers are principals (or more specifically sids) who may have contributed to the generation of the secret, whereas the readers are the principals (sids) that are meant to read (and hence use) the secret. Informally, if one of the issuers of a labeled secret is corrupted, then the secret may have been chosen by the adversary. Conversely, if one of the readers is corrupted, then the secret may eventually be leaked to the adversary. If both the issuers and readers remain uncorrupted, we expect that the secret should remain confidential.

Sending and Receiving Network Messages. The event SendMsg s r m simulates a network message m sent from principal s to r. Since our threat model considers active network attackers, we provide the adversary with functions to read any message in the trace and to inject new messages from any s to any r. An honest principal r can only read messages addressed to it, but by injecting new messages, the attacker can control which messages it receives.

B. A Typed Symbolic Cryptographic Interface

The messaging protocols in this paper rely on a cryptographic library that is concretely implemented using cryptographic algorithms that operate over bytestrings. However, to precisely state our cryptographic assumptions about these algorithms, we replace this library with a *symbolic* implementation of cryptography in F* that operates over abstract terms. We then prove that this symbolic model meets a typed cryptographic interface that reflects our assumptions about functionality and security of each primitive. This modeling style is similar to prior work [13], [14], but adapted to work with our model of global execution and state compromise.

Labeled Key Material. The typed cryptographic interface tracks the label l and usage u of any secret key material that is generated or derived via some cryptographic function. A public key pk is said to have a label l and usage u to mean that the private key sk corresponding to pk is a secret with label l and usage u. We say that a label l is *stricter* than a label l' if all the issuers (resp. readers) in l' are included in the issuers (resp. readers) of l. In particular, every label is stricter than Public. We say that a label l is *corrupted* if one of the issuers or readers in l is covered by a Corrupt event in the global trace. A corrupted label is less strict than Public.

We use labeling to control the flow of information as it passes through the cryptographic library, session storage, and the public network. Informally, data can flow from less strict to more strict labels. Only data that is less strict than Public should be sent on the network (since it will become visible to the adversary). Only secrets whose label includes a principal's session should be stored in the corresponding session state.

Labeling Rules for Cryptographic Functions. For every cryptographic function in the library, the interface specifies a type that indicates labeling pre-conditions and post-conditions. TreeKEM$_B$ uses functions for transcript hashing, key derivation, multi-recipient public-key encryption, AEAD encryption, and public-key signatures. Each of these functions requires that the key material provided has the right usage, and the payloads have the right labels. For example, mpke_enc s eks takes a secret with label l, and an array of public keys eks, where each public key in the array has usage PKE and a label that is stricter than l, and it returns a public ciphertext. The key derivation function kdf_derive s takes a secret s with label l and usage KDF (l',u') u'', where l is stricter than l', and returns a secret that has label l' and usage u'.

The rule for signatures is a bit different: sign sk m takes a secret sk with label l and usage Sig, and a message m that satisfies some application-specific predicate sig_pred l m, and returns a public signature. Conversely, if verify vk m sg succeeds for a message m and a verification key with an uncorrupted label l, then it guarantees that sign_pred l m holds.

In addition to these security assumptions, the interface also provides correctness lemmas stating, for example, that if the keys match, then decryption is an inverse of encryption.

Attacker API and Secrecy Lemma. In our model, the attacker is an F⋆ program that is given an API allowing it to exercise a wide range of capabilities. It can read and write messages to the network, it can read the state of compromised principals, it can generate its own random values, and it can call any cryptographic function. It it can also call functions in the stateful application API provided by TreeKEM$_B$, hence triggering any group operation at any principal.

The main limitation we place on the attacker is that it can only learn secrets via its given interface. It cannot guess randomly generated bytestrings or break the abstractions provided by our symbolic cryptographic model.

The attacker's knowledge is encoded as a monotonic predicate over the global trace; that is, as the trace grows, so does the attacker's knowledge. To prove the soundness of our labeled cryptographic interface, we prove a general *symbolic secrecy lemma* for all well-typed protocols that use our cryptographic interface stating that secrets with label l can only be known to the adversary after l has been corrupted.

C. Verifying the Security of TreeKEM$_B$ in F⋆

To prove the security of TreeKEM$_B$, we first define a *tree invariant* that must be preserved by all the group management functions in our specification. We then define a *signature predicate* that must hold whenever a sender signs a message. Then, relying on both of these predicates, we show how to obtain the security goals of MLS.

Tree Invariant. For each tree generated in TreeKEM, we compute a *tree label*, by taking the union of the auth sessions of all its members as the set of issuers, and the union of the dec sessions of all its members at the set of readers. This tree label represents the secrecy level of the current membership of the tree; if any of its issuers is compromised, then the tree may have a malicious insider; if any of its readers is corrupted, then the tree's subgroup secret may be leaked to the adversary. We can then state our security invariant for TreeKEM$_B$ as an invariant on each subtree of the group state:

- Every occupied leaf in the tree with member_info mi contains a valid credential with a verification key labeled with the auth session of mi. Further, the current encryption key at the leaf is labeled with the dec session of mi.
- Every non-blank node in the tree contains a key package with a public encryption key and a ciphertext. If none of the members of the sub-tree are auth_compromised, then the label of the encryption key matches the tree label of the subtree, and the ciphertext contains an encrypted secret that is also labeled with the tree label of the subtree.

By typechecking in F⋆, we prove that this invariant holds in create, and is preserved when a sender locally applies an operation it generates by calling modify; that is, it is preserved by both blank_path and update_path. The proof of each lemma is by induction and case analysis over the tree structure.

Next, we need to prove that the invariant holds at recipients after they receive a Create, Welcome, or Modify message, but for this, we need to rely on the sender's signature, and hence on the TreeKEM$_B$ signature predicate.

Signature Predicate. We require that every signature produced by an honest member in TreeKEM$_B$ must satisfy one of the following conditions:

- the signed message is a Create or Welcome and contains the sender's group state which satisfies the tree invariant,
- the signed message is a Modify, and the resulting group state at the sender satisfies the tree invariant, or
- the signed message is a AppMsg, and is a message that the sender intended to send in the current group state.

We first prove that this predicate holds at all calls to sign in our TreeKEM$_B$ specification. We then try to prove that after a recipient processes a signed message, the subsequent state satisfies the tree invariant. This indeed holds for the Create but fails for Welcome. This is because the recipient of a Welcome message is a new member who cannot verify the tree invariant itself, and so it has to rely on the honesty of the sender. As we shall see in Section VI, this proof failure exposes the recipient to a variation of the Double Join attack. Once we adopt the mitigation for the attack in TreeKEM$_{B+S}$, we show that recipients of Welcome messages do obtain the invariant.

Next, we try to prove that the recipient of Modify and AppMsg agrees with the sender on the group state. This proof fails again, because it turns out that the signature for these messages does not include the transcript hash or any other unique representative of the group state (see Figure 6). This proof failure results in a cross-group forwarding attack, described in Section VI. Once we fix the protocol to include the transcript hash in all signatures, we are able to prove both the secrecy invariant and group agreement after all messages.

Proving MLS Security. By applying the subgroup secrecy invariant to the root secret, and then to the derived messaging keys, we can show that these keys are labeled with the current members of the group. By then applying the symbolic secrecy lemma, we can prove that these keys, and hence the application messages they encrypt are confidential as long as the current membership is uncompromised. This is enough to obtain Msg-Conf, Add-FS, Rem-PCS, Upd-FS, and Upd-PCS. By additionally relying on the signature predicate for application messages, we obtain Msg-Auth and Grp-Agr.

VI. Attacks and Mitigations for MLS Draft 7

Our formal proofs of TreeKEM$_B$ uncovered two attacks on the protocol: one violates the subgroup secrecy invariant after Welcome, and the other violates group agreement after Modify.

Double Join Attack on New Members. When a new member b receives a group state in a Welcome message from some old member a, it has no idea if the tree is valid: if a is malicious it could attempt an active *double join attack* by sending a tree with its own public keys at all leaves and non-blank nodes. In the resulting tree, the public keys do not correspond to the membership of the subtrees, violating the tree invariant.

This invariant violation may result in many attacks; we show a concrete attack on the Rem-PCS property in Figure 8. Suppose a malicious insider a adds f but puts the wrong public

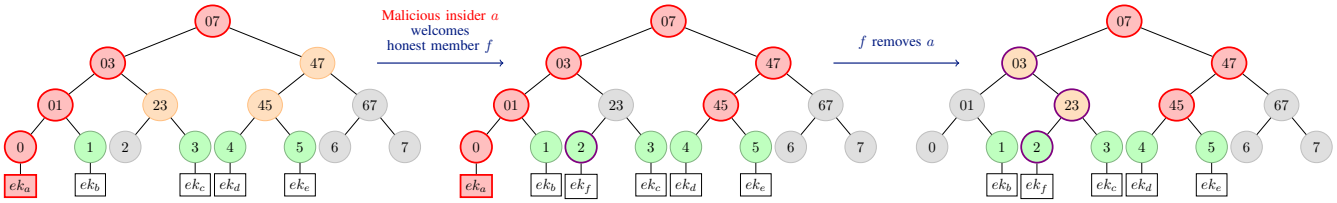

Fig. 8. Double Join Attack on new members in TreeKEM$_{\text{B}}$. A malicious (actively compromised) member a (at index 0) adds a new member f (at index 2). As expected, it blanks the path from 2 to the root, then generates an update from 0 to the root. However, when sending the Welcome message to f, a replaces the encryption keys for 47 and 45 with its own encryption keys. Now, even if a is removed by f, it can still compute the new group secret s_{07}.

keys in the tree; suppose then that f removes a, and sends a message m to the new group. At this point, a is not in the group any more, and we expect that m cannot be read by a but this is not the case, since the group secret is still known to a. In other words, removing a did not heal the group state at f, breaking the expected Remove Security guarantee (Rem-PCS).

Fixing this attack is not easy, since it is inherent in a protocol where any member can add any other. If we restricted the add capability to a trusted set of members, then we can ignore this invariant violation by treating these trusted members as *implicit members* of every subtree. However, such a design requires a significant amount of trust in some principals, which is antithetical to the principles of MLS.

TreeKEM$_{\text{B+S}}$: Signed Trees for TreeKEM$_{\text{B}}$. We propose to extend TreeKEM$_{\text{B}}$ so that the group-state becomes self-authenticating. The key package in every non-blank node of the tree is extended with a signature over (a hash of) the contents of the subtree by the last actor to have modified the subtree, who must be one of the members of the subtree. This prevents a malicious insider from tampering with any subtree it is not a member of, hence restoring the tree invariant.

To initialize the tree, the creator adds a signature to each node on the path from the creator's leaf to the root. Recall that the initial leaf encryption keys are already signed by the members at the leaves. (All other nodes in the initial tree are blank.) Similarly, at each Modify the sender adds a signature to each node on the updated path. To Welcome a new member, the sender forwards the full signed tree to the new member.

When receiving a message, the recipient follows the same rules as TreeKEM$_{\text{B}}$; in particular it does not have to verify the signatures. The only exception is that the new member who receives a Welcome should verify all the signatures (especially if it does not trust the sender). Hence, the only change in complexity is this $O(n)$ computation at new members.

We have modeled a simple version of TreeKEM$_{\text{B+S}}$ and shown that it satisfies our security goals, fixing the attack on Rem-PCS. We have proposed this fix to the MLS working group and early indications are that it will be accepted into the protocol. We plan to extend our model with a precise proof of TreeKEM$_{\text{B+S}}$ once all the low-level details of this proposal have been concretely specified in a future MLS draft.

Cross-Group Forwarding Attack. When a group member b receives a message over a group g, it needs to know that the sender a intended to send the message to this group. If the sender's signature does not include the group state or transcript hash, the recipient does not get this guarantee, leading to a cross-group forwarding attack.

Suppose a sender a and an attacker b both belong to two groups g_1 and g_2. If a sends a message on g_1, b can decrypt it, and then re-encrypt it and send it on g_2 even though a never intended to send this signed message on g_2. To honest recipients on g_2 it will appear as if a is responding to their conversation, which is not the case. To make this attack work in practice requires other pre-conditions. Both groups must have the same identifier gid for the signed header to be valid on both groups. Since these groups may well be served by independent delivery servers, we believe this is plausible.

The fix for this attack is easy: the transcript hash should be added to all signatures. We have proposed this fix to the MLS working group and it will be incorporated in the next draft.

Other MLS Weaknesses and Improvements. During the course of this work, we (and others) identified several other weaknesses in the MLS and communicated them to the working group. As we pointed out in Section IV, Create and Welcome messages were unsigned in draft 7, which we think was an oversight by the protocol authors. We also found a truncation attack that a network attacker can exploit between MLS epochs.

Stream Truncation : In MLS draft 7 (and in our model), each application message has an authenticated counter called *generation* that is reset to zero in every new epoch. A network attacker can eagerly drop application messages belonging to the old epoch when the epoch changes, causing the stream of messages at the sender and recipients to be inconsistent, or stream truncation. This attack can be fixed by adding a previous_counter field to each application message, as is done in Signal.

It is also important to note that the FS and PCS guarantees of TreeKEM$_{\text{B}}$ (and TreeKEM$_{\text{B+S}}$) rely crucially on each member of the group regularly updating its decryption keys (for secrecy) and credentials (for authenticity). In TreeKEM$_{\text{B}}$, even the efficiency of the protocol relies on members regularly sending updates in order to un-blank nodes. In our model, we do not specify any frequency for these updates, leaving this decision to the messaging application. In parallel work to this paper, other researchers have been looking at designs that try to improve the FS and PCS guarantees of TreeKEM$_{\text{B}}$ [24], [25]. These proposals are currently being discussed in the working group, and we believe that they can and should also be formally evaluated using a comprehensive formal model like ours.

Finally, in our work, we identified some efficiency improvements for Add and Remove that have now being discussed for incorporation into the protocol. For Add, we recommend that,

instead of blanking the path from the new member to the root, the encryption key of the new member be added as an extra encryption key for each node on the path. For Remove, we recommend that the sender send an update *after* removing the member, not before as in draft 5, hence un-blanking some of the nodes on the path from the deleted leaf to the root. Both these changes significantly reduce the cost of subsequent group operations by minimizing the number of blank nodes in the tree without changing the essence of the protocol. The change we proposed to Remove was incorporated into draft 6, and the change to Add is under discussion.

VII. Related Work and Conclusions

Unger *et al* [3] survey messaging protocols, and point out the paucity of group messaging designs with strong security guarantees, compared to the wealth of work on attacks [26], [27], [28], [29], [30], security definitions [31], [32], [33], [34] and verified protocols [35], [36], [30] for two-party messaging.

Rösler *et al* [37] describe several vulnerabilities in the way group chats are implemented in popular messaging applications, indicating the need for formal analysis. ART was the first protocol to support asynchronous group messaging with FS and PCS [7]. However, the analysis of ART does not account for Remove, sender authentication, or malicious insiders. Our machine-checked model generalizes ART and accounts for all of these. In parallel to our work, Alwen *et al* [24] observe that the FS guarantees of TreeKEM$_B$ could be improved by using *updatable* public key encryption, whereas Cremers *et al* [25] focus on improving the PCS guarantees of TreeKEM$_B$. Both these extensions are under discussion at the MLS working group and we plan to extend our model to verify their designs.

We present a new model of messaging in F* that allows for unbounded size groups, recursive stateful data structures, and fine-grained compromise. This allows us to prove FS and PCS guarantees for the first time in F*, and to find concrete attacks that would not be visible in a more abstract model of MLS.

This work represents two years of engagement with the MLS working group. Our work has already had a significant impact. Our preliminary analyses influenced the design of both TreeKEM and TreeKEM$_B$. We presented our attacks to the working group and our proposed fixes are being incorporated into the next draft. We intend to continue to develop our model and track the standard as it evolves.

References

[1] M. Marlinspike and T. Perrin, "Signal protocol," 2016, https://signal.org/docs.

[2] Telegram, "Mtproto mobile protocol version 2.0: End-to-end encryption, secret chats," 2017, https://core.telegram.org/api/end-to-end.

[3] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "Sok: Secure messaging," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015, pp. 232–249.

[4] M. Marlinspike and T. Perrin, "The x3dh key agreement protocol," 2016, https://signal.org/docs/specifications/x3dh/.

[5] T. Perrin and M. Marlinspike, "The double ratchet algorithm," 2016, https://signal.org/docs/specifications/doubleratchet/.

[6] I. Levy and C. Robinson, "Principles for a more informed exceptional access debate," https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate, 2018.

[7] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1802–1819.

[8] H. Liu, E. Y. Vasserman, and N. Hopper, "Improved group off-the-record messaging," in *ACM Workshop on Workshop on Privacy in the Electronic Society (WPES)*, 2013, pp. 249–254.

[9] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups," May 2018, published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8. [Online]. Available: http://prosecco.inria.fr/personal/karthik/pubs/treekem.pdf

[10] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 256–270.

[11] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of TLS 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1773–1788.

[12] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 483–502.

[13] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, 2010, pp. 445–456.

[14] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, 2011.

[15] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[16] D. A. Basin, C. Cremers, J. Dreier, and R. Sasse, "Symbolically analyzing security protocols using tamarin," *SIGLOG News*, vol. 4, no. 4, pp. 19–30, 2017.

[17] E. Omara, B. Beurdouche, E. Rescorla, S. Inguva, A. Kwon, and A.Duric, "Messaging layer security architecture," IETF Inernet Draft, February 2018.

[18] M. Marlinspike, "Private group messaging," 2014, https://signal.org/blog/private-groups/.

[19] N. P. Smart, *Efficient Key Encapsulation to Multiple Parties*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 208–219. [Online]. Available: https://doi.org/10.1007/978-3-540-30598-9_15

[20] Y. Kim, A. Perrig, and G. Tsudik, "Tree-based group key agreement," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 60–96, Feb. 2004.

[21] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[22] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 445–459.

[23] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer." IEEE Computer Society, 2017, pp. 463–482.

[24] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the ietf mls standard for group messaging," Cryptology ePrint Archive, Report 2019/1189, 2019, https://eprint.iacr.org/2019/1189.

[25] C. Cremers, B. Hale, and K. Kohbrok, "Revisiting post-compromise security guarantees in group messaging," Cryptology ePrint Archive, Report 2019/477, 2019, https://eprint.iacr.org/2019/477.

[26] M. D. Raimondo, R. Gennaro, and H. Krawczyk, "Secure off-the-record messaging," in *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2005, pp. 81–89.

[27] S. R. Verschoor and T. Lange, "(in-)secure messaging with the silent circle instant messaging protocol," *IACR Cryptology ePrint Archive*, vol. 2016, p. 703, 2016. [Online]. Available: http://eprint.iacr.org/2016/703

[28] A. Rad and J. Rizzo, "A 2^{64} attack on Telegram, and why a super villain doesn't need it to read your telegram chats." 2015.

[29] J. Jakobsen and C. Orlandi, "On the CCA (in)security of mtproto," in *Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2016, pp. 113–116.

[30] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic

and computational approach," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 435–450.

[31] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[32] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *Advances in Cryptology – CRYPTO 2017*, 2017, pp. 619–650.

[33] B. Poettering and P. Rösler, "Towards bidirectional ratcheted key exchange," in *Advances in Cryptology – CRYPTO 2018*, 2018, pp. 3–32.

[34] A. Lehmann and B. Tackmann, "Updatable encryption with post-compromise security," in *Advances in Cryptology – EUROCRYPT 2018*, 2018, pp. 685–716.

[35] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is TextSecure?" in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[36] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[37] P. Rösler, C. Mainka, and J. Schwenk, "More is less: On the end-to-end security of group chats in signal, whatsapp, and threema," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 415–429.

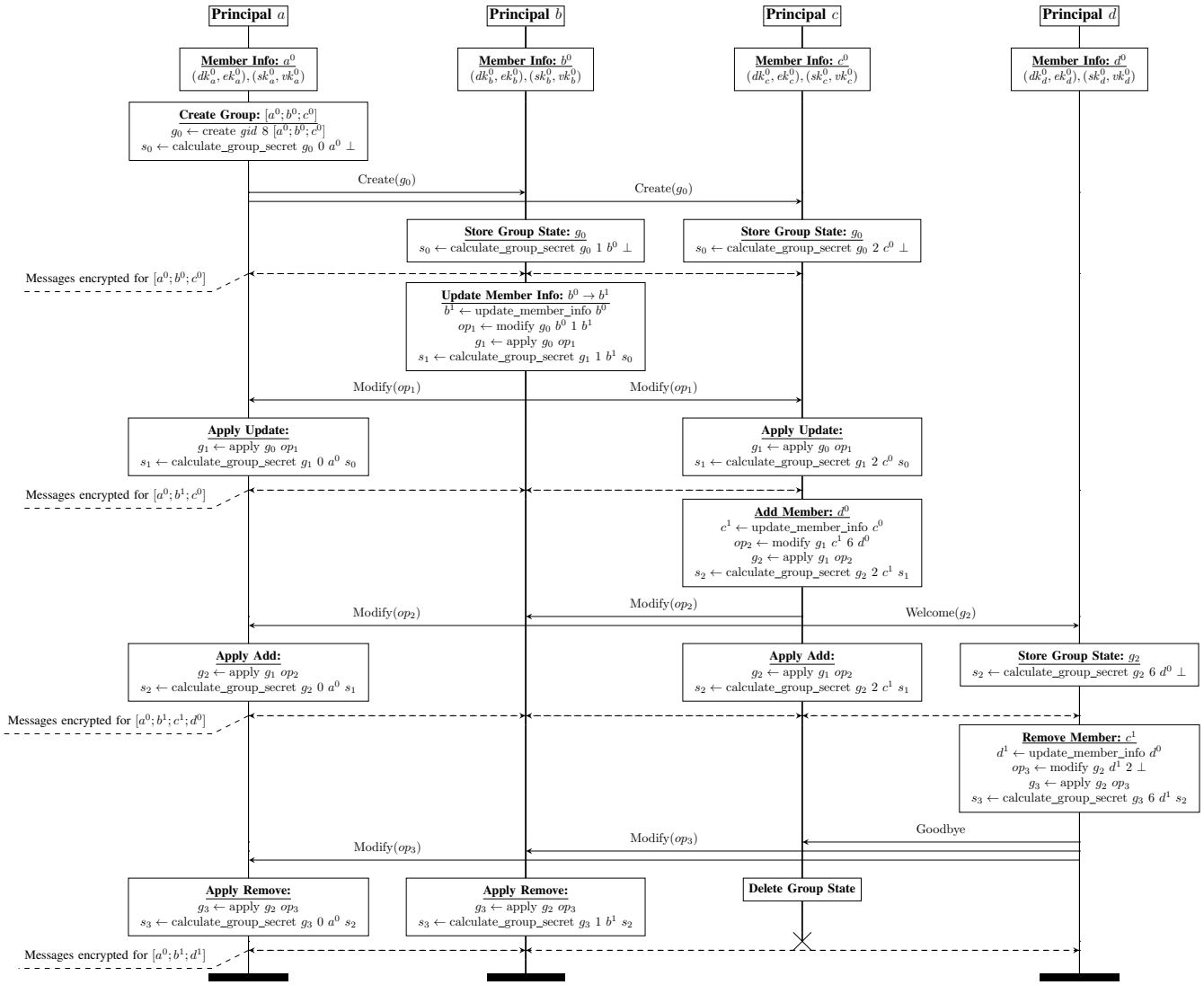

Fig. 9. Evolution of a Group: (1) a creates a group with three members $\{a, b, c\}$; (2) b updates its keys; (3) c adds d to the group; (4) d removes c. Once its local group state is initialized, a member can securely send messages to the group at any time, by encrypting it using the current group state.

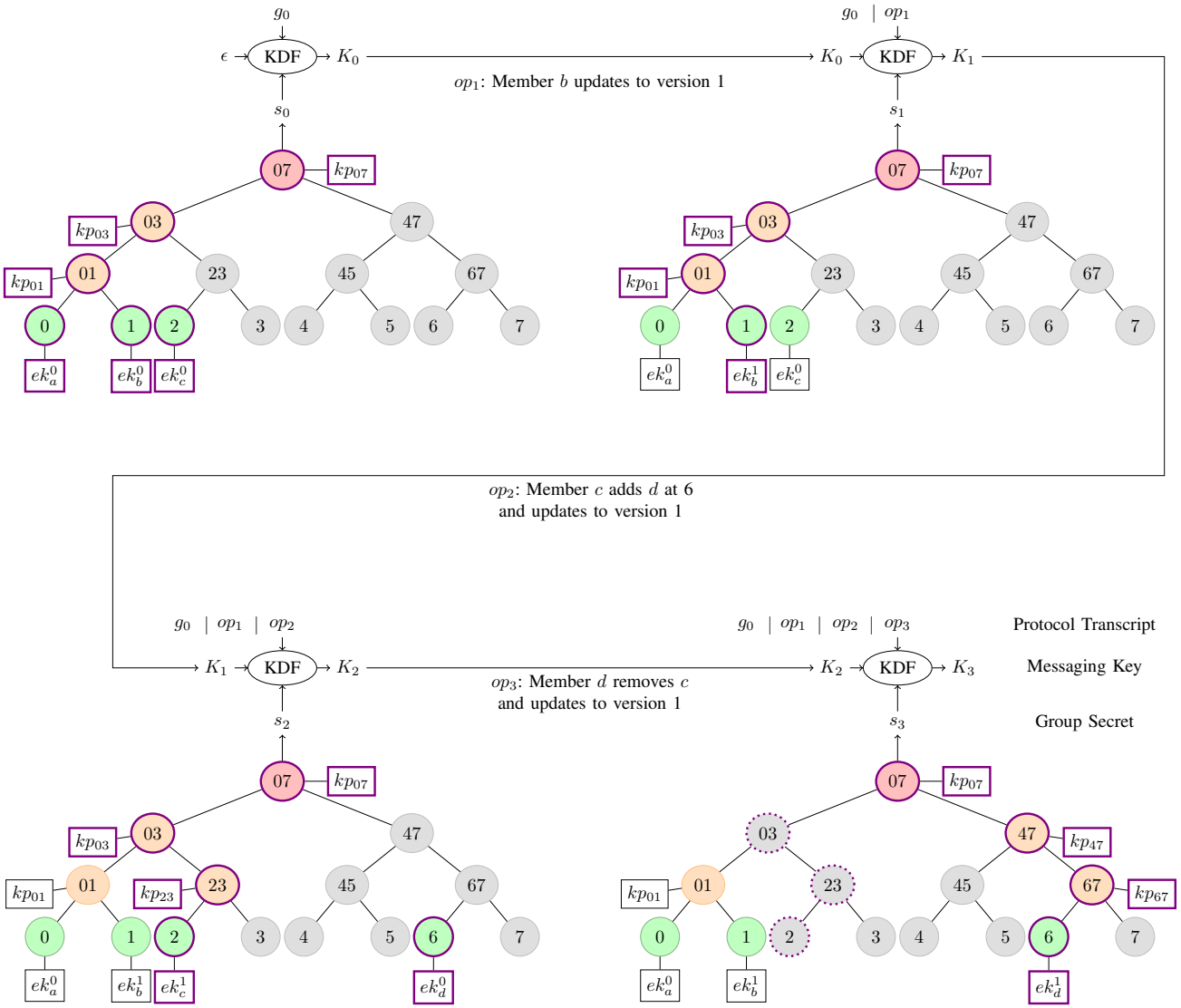

Fig. 10. Evolution of a Group: (1) a creates a group g_0 with three members $\{a, b, c\}$; (2) b updates its keys; (3) c adds d to the group; (4) d removes c. Once its local group state is initialized, a member can securely send messages to the group at any time, by encrypting it using the current group state. This describes the same scenario as in figure 9. Plain violet borders indicates that a value was just updated, dotted borders indicates a value that was just blanked.

```
(* Definition of the main data structures in TreeKEM_B *)
type member_secrets = {
      identity_sig_key: sign_key;
      leaf_secret:secret;
      current_dec_key: dec_key}
type direction = | Left | Right
type key_package = {
      from : direction;
      node_enc_key: enc_key;
      node_ciphertext: bytes}
type group_state = {
      group_id: nat;
      levels: nat;
      tree: tree levels;
      epoch: nat;
      transcript_hash: bytes}
let index_l (l:nat) = x:nat{x < pow2 l}
type operation = {
      lev: nat;
      index: index_l lev;
      actor: credential;
      path: path lev & path lev}
type group_secret :eqtype = {
      init_secret: secret; hs_secret: secret; sd_secret: secret;
      app_secret: secret; app_generation: nat}
```

```
(* Auxiliary tree λctions *)
let child_index (l:pos) (i:index_l l) : index_l (l−1) & direction =
  if i < pow2 (l − 1) then (i,Left) else (i−pow2 (l−1),Right)
let key_index (l:nat) (i:index_l l) (sib:tree l) dir : index_l (l+1) =
  if dir = Left then i else i + length (pub_keys l sib)
let order_subtrees dir (l,r) = if dir = Left then (l,r) else (r,l)
```

```
(* Create a new tree from a member array *)
let rec create_tree (l:nat) (c:credential)
                    (init:member_array (pow2 l)) =
  if l = 0 then Leaf c init.[0]
  else let init_l,init_r = split init (pow2 (l−1)) in
       let left = create_tree (l−1) c init_l in
       let right = create_tree (l−1) c init_r in
       Node c None left right
(* Apply a path to a tree *)
let rec apply_path (l:nat) (i:nat{i<pow2 l}) (a:credential)
                   (t:tree l) (p:path l) : tree l =
  match t,p with
  | Leaf _ m, PLeaf m' → Leaf a m'
  | Node _ _ left right, PNode nk next →
    let (j,dir) = child_index l i in
    if dir = Left
    then Node a nk (apply_path (l−1) j a left next) right
    else Node a nk left (apply_path (l−1) j a right next)
(* Create a blank path after modifying a leaf *)
let rec blank_path (l:nat) (i:index_l l)
                   (mi:option member_info) : path l =
  if l = 0 then PLeaf mi
  else let (j,dir) = child_index l i in
       PNode None (blank_path (l−1) j mi)
(* Create an update path from a leaf to the root *)
let rec update_path (l:nat) (t:tree l) (i:nat{i<pow2 l})
                    (mi_i:member_info) (s_i:secret)
                    : option (path l & s_root:secret) =
  match t with
  | Leaf _ None → None
  | Leaf _ (Some mi) → if name(mi.cred) = name(mi_i.cred)
                       then Some (PLeaf (Some mi_i),s_i)
                       else None
  | Node _ _ left right →
    let (j,dir) = child_index l i in
    let (child,sibling) = order_subtrees dir (left,right) in
    match update_path (l−1) child j mi_i s_i with
      | None → None
      | Some (next, cs) →
        let ek_sibling = pub_keys (l−1) sibling in
        let kp,ns = node_encap cs dir ek_sibling in
        Some (PNode (Some kp) next, ns)
```

```
(* Create a new group state *)
let create gid sz init leaf_secret =
  match init.[0], log2 sz with
  | None,_ → None
  | _,None → None
  | Some c,Some l →
    let t = create_tree l c.cred init in
    let ek = pk leaf_secret in
    let mi' = {cred=c.cred; version=1; current_enc_key=ek} in
    (match update_path l t 0 mi' leaf_secret with
    | None → None
    | Some (p,_) → let t' = apply_path l 0 c.cred t p in
                   let g0 = {group_id=gid; levels=l;
                             tree=t'; epoch=0;
                             transcript_hash = empty} in
                   let h0 = hash_state g0 in
                   Some ({g0 with transcript_hash = h0}))
(* Apply an operation to a group state *)
let apply g o =
  if o.lev ≠g.levels then None
  else let p1,p2 = o.path in
       let t' = apply_path o.lev o.index o.actor g.tree p1 in
       let t' = apply_path o.lev o.index o.actor g.tree p2 in
       Some ({g with epoch = g.epoch + 1; tree = t';
              transcript_hash = hash_op g.transcript_hash o})
(* Create an operation that modifies the group state *)
let modify g actor i mi_i leaf_secret =
    match (membership g).[actor] with
    | None → None
    | Some mi_a_old →
      let mi_a = update_member_info mi_a_old leaf_secret in
      let bp = blank_path g.levels i mi_i in
      let nt = apply_path g.levels i mi_a.cred g.tree bp in
      match update_path g.levels nt actor mi_a leaf_secret with
      | None → None
      | Some (up,_) → Some ({lev = g.levels; actor= mi_a.cred;
                             index = i; path = (bp,up)})
```

```
(* Calculate the subgroup secret for the root of a tree *)
let rec root_secret (l:nat) (t:tree l) (i:index_l l) (leaf_secret:bytes)
        : option (secret & j:nat{j < length (pub_keys l t)}) =
  match t with
  | Leaf _ None → None
  | Leaf _ (Some mi) → Some (leaf_secret, 0)
  | Node _ (Some kp) left right →
    let (j,dir) = child_index l i in
    let (child,_) = order_subtrees dir (left,right) in
    (match root_secret (l−1) child j leaf_secret with
    | None → None
    | Some (cs,i_cs) →
      let (_,recipients) = order_subtrees kp.from (left,right) in
      let ek_r = pub_keys (l−1) recipients in
      (match node_decap cs i_cs dir kp ek_r with
        | Some k → Some (k,0)
        | None → None))
  | Node _ None left right →
    let (j,dir) = child_index l i in
    let (child,sib) = order_subtrees dir (left,right) in
    match root_secret (l−1) child j leaf_secret with
    | None → None
    | Some (cs,i_cs) → Some (cs,key_index (l−1) i_cs sib dir)
(* Calculate the current group secret *)
let calculate_group_secret g i ms gs =
  match root_secret g.levels g.tree i ms.leaf_secret with
  | None → None
  | Some (rs,_) →
    let prev_is = if gs = None then empty_bytes
                  else (Some?.v gs).init_secret in
    let (apps,hs,sds,is') =
        derive_epoch_secrets prev_is rs g.transcript_hash in
    Some ({init_secret = is';hs_secret = hs; sd_secret = sds;
           app_secret = apps; app_generation = 0})
```

Fig. 11. An F* specification of TreeKEM$_B$, implementing the group management and key exchange functions in the MLS interface of Figure 2. Our full reference implemention includes code for caching subgroup secrets, parsing and serializing groups and operations, message protection, and a stateful API.