TreeSync: Authenticated Group Management for Messaging Layer Security

Théophile Wallez
Inria Paris

Jonathan Protzenko
Microsoft Research

Benjamin Beurdouche
Mozilla

Karthikeyan Bhargavan
Inria Paris

Abstract

Messaging Layer Security (MLS), currently undergoing standardization at the IETF, is an asynchronous group messaging protocol that aims to be efficient for large dynamic groups, while providing strong guarantees like forward secrecy (FS) and post-compromise security (PCS). While prior work on MLS has extensively studied its group key establishment component (called TreeKEM), many flaws in early designs of MLS have stemmed from its group integrity and authentication mechanisms that are not as well-understood. In this work, we identify and formalize TreeSync: a sub-protocol of MLS that specifies the shared group state, defines group management operations, and ensures consistency, integrity, and authentication for the group state across all members.

We present a precise, executable, machine-checked formal specification of TreeSync, and show how it can be composed with other components to implement the full MLS protocol. Our specification is written in F* and serves as a reference implementation of MLS; it passes the RFC test vectors and is interoperable with other MLS implementations. Using the DY* symbolic protocol analysis framework, we formalize and prove the integrity and authentication guarantees of TreeSync, under minimal security assumptions on the rest of MLS. Our analysis identifies a new attack and we propose several changes that have been incorporated in the latest MLS draft. Ours is the first testable, machine-checked, formal specification for MLS, and should be of interest to both developers and researchers interested in this upcoming standard.

1 Introduction

Whether WhatsApp, Signal, Facebook Messenger, or Wire, virtually all modern messaging applications prominently advertise end-to-end encryption (E2EE) as one of their core features, confirming that private, secure communications are becoming a baseline expectation for many users. Unlike short-lived HTTPS connections, however, messaging conversations can run for years, and so the security guarantees of messaging must account for the realistic possibility that one of the devices is stolen or otherwise compromised during the lifetime of the conversation. If an adversary compromises a device, it can of course read recent messages and send new messages, but we would still like to protect messages that were sent or received well in the past, i.e. *forward secrecy* (FS), and messages that will be sent or received in the future after a period of healing, i.e. *post-compromise security* (PCS).

For two-party conversations, messaging applications rely on modern protocols like Signal [34] to provide FS and PCS by regularly updating (or *ratcheting*) the message encryption keys [36]. However, many messaging conversations involve *groups* of more than two parties. Indeed, since many users have several devices, even a chat between two individuals becomes a group conversation under the hood.

Group Messaging. The fundamental difference between group messaging and two-party conversations is that groups are dynamic: participants may enter or leave at any time, meaning that the membership (or *roster*) evolves over time. The message encryption keys must also evolve with changes in the roster, so that, for example, a member who has been removed from the group cannot read subsequent messages. To keep track of the group membership, each member needs to continuously synchronize and authenticate the current group state, so they know who they are talking to.

The security requirements for group messaging are also more complex: confidentiality properties like FS and PCS need to be adapted for groups where any member device may be compromised, and authentication guarantees must now also include group membership authentication and sender authentication within the group. Groups can also grow quite large, so a group messaging protocol must provide all these guarantees while scaling to a roster with up to thousands of members. See [44] for a detailed discussion on the challenges of group messaging and a survey of proposed designs.

Current group messaging applications meet a subsets of these requirements. For example, WhatsApp uses the Signal Sender Keys protocol [32] which uses two-party Signal chan-

nels between each pair of members to distribute keys for group conversations. This protocol provides FS and sender authentication, but does not authenticate group membership, does not provide PCS, and its reliance on n^2 Signal channels in groups of size n does not scale to large groups. Signal recently added a private group system [22] that adds membership authentication and privacy but does not improve efficiency.

To address this state of affairs, the IETF has convened a working group tasked with designing a new secure group messaging protocol, dubbed MLS (Messaging Layer Security). MLS is nearing publication, and an early implementation is already deployed in RingCentral and Cisco's Webex video conferencing platform. The security of MLS is of great interest, as it is likely to be adopted by several messaging applications.

IETF MLS. The IETF MLS working group is tasked with designing a protocol that achieves the goals described in the MLS architecture [12]. The architecture assumes the existence of a trusted authentication service (AS), which attests to relationships between member *identities* and their authentication *credentials*. It also assumes the existence of a mostly-untrusted delivery service (DS), which stores and delivers messages to endpoints and defines a globally unique order for all group modifications. A malicious DS may ignore some messages, or partition the group by selectively delivering messages, but it cannot read or write group messages.

The MLS protocol is currently at version 16 [8] and is in the final stages of standardization. The first few drafts of the MLS protocol were primarily concerned with efficiently establishing group keys for large groups. The starting point was Asynchronous Ratcheting Trees (ART) [23], a protocol that uses a tree of Diffie-Hellman keys to efficiently update and distribute group keys, providing both FS and PCS. ART was replaced (in draft 2) by a more efficient alternative called TreeKEM [13] that is based on Hybrid Public Key Encryption [10]. Subsequent drafts refined and improved TreeKEM, but the fundamental key establishment mechanism remains the same. The TreeKEM protocol (and many of its variants) have been formally analyzed in the literature [3–5, 21].

The group key established by TreeKEM is then used to derive a tree of message encryption keys that each group member can use to send and receive application messages, via a protocol we call TreeDEM (introduced in draft 7).

Both TreeKEM and TreeDEM rely on a group state data structure that must be synchronized across all current members. Most of the remaining complexity of MLS is in defining this data structure, specifying how members can modify the group state to add and remove members, and how the group state is synchronized and authenticated between members. Indeed, many of the recent significant changes in the protocol have been motivated by strengthening the integrity and authentication guarantees of the group state against insider and outsider attacks. For example, an early attack called *double join* allowed a member to resist future removal by surreptitiously adding itself to the group. Avoiding this attack resulted in significant changes to the treatment of the member removal, at the cost of making TreeKEM less efficient. More recent authentication attacks on new members [5, 14] motivated the design of a complex *parent hash* mechanism to protect the integrity of the group state. Despite these attacks and resulting changes, the authentication mechanisms of MLS have not been studied in their own right, and prior works have primarily seen group management from the narrow lens of its impact on key establishment in TreeKEM.

Contributions. In this paper, we focus specifically on the group state management and authentication mechanisms of MLS, which we identify as a separate sub-protocol called TreeSync. We show that MLS can be cleanly decomposed into TreeKEM, TreeDEM, and TreeSync, allowing us to state and prove the authentication and integrity guarantees of TreeSync independently of TreeKEM and TreeDEM.

We present a machine-checked formal security analysis of a byte-level precise specification of TreeSync written in the F* programming language [43]. Our specification is executable and serves as a reference implementation of MLS, which we test and evaluate against other implementations.

Our analysis uncovers a new attack that exploits the interaction between TreeSync and TreeDEM, and also highlights other issues in MLS. We propose fixes for these issues, which have been incorporated into MLS.

We prove a series of integrity and authentication theorems for TreeSync in MLS draft 16, using the DY* symbolic protocol analysis framework [15]. Notably, our proofs make no assumptions on the security of TreeKEM, and we only need minimal assumptions on the use of signatures in TreeDEM.

Ours is the first testable, machine-checked, formal specification for MLS. It covers all details of the protocol down to the precise message formats, and hence may be of independent interest to developers and researchers interested in MLS. Conversely, our proofs are only for TreeSync; although we formally specify both TreeKEM and TreeDEM, we leave their comprehensive security analysis for future work.

Outline. We start with a new presentation of MLS as the combination of three independent subsystems (§2). We then turn our attention to one of those, TreeSync, and precisely capture its behavior (§3). Equipped with the specification, we then set out to formally prove the security of TreeSync in the symbolic model (§4). Our contribution is not purely theoretical: our implementation is usable, interoperable, and has been successfully integrated in a prototype version of the Skype messaging client (§5). Our proof and implementation combined have influenced both the standard and other implementations: we describe changes to the MLS draft that resulted from attacks we found, as well as bugs in other implementations that were exposed through our work (§6). We conclude with related work (§7). Our verified implementation is available online as part of the anonymous supplement [1].

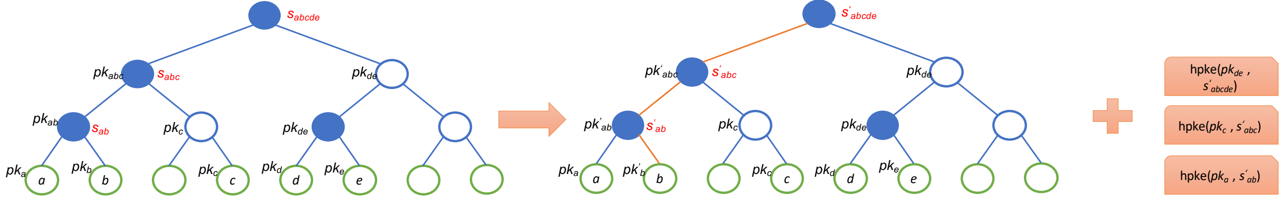

Figure 1: **TreeKEM** maintains a tree of subgroups, each associated with a secret (e.g. s_{abc}) and corresponding public key (pk_{abc}). The root secret (s_{abcde}) is the commit secret for the current epoch. Each member (e.g. b) only knows the secret keys for the subgroups it is a member of ($s_{ab}, s_{abc}, s_{abcde}$). To send a commit, a member (e.g. b) updates its subgroup secrets (to $s'_{ab}, s'_{abc}, s'_{abcde}$) and encrypts each new subgroup secret (e.g. s'_{abc}) to the corresponding sibling subgroup's public key ($hpke(pk_c, s'_{abc})$). Hence, each commit for a group of n (e.g. 8) members results in only $\log(n)$ $(= 3)$ public key encryptions.

2 MLS: TreeKEM, TreeDEM, and TreeSync

The Messaging Layer Security (MLS) protocol [8] enables a set of endpoints to form a dynamic group and exchange end-to-end encrypted messages that only the current members of the group can read or write. We begin with a high-level view of this protocol before describing its cryptographic components.

Dynamic Groups. To initiate a group conversation, an endpoint, called the *creator*, creates a new group and assigns it a fresh group secret. The creator can then add other members to this group by sending them encrypted *welcome* messages containing group information, including the current membership and the group secret. Each member is authenticated by a *credential* issued by a trusted Authentication Service, which associates the member with a signature keypair.

Any member of the group can subsequently *propose* to add or remove members, or update its own credential and/or encryption keys. A group member can *commit* a batch of pending proposals by modifying the group, updating the group secret, and conveying the new secret to the updated set of group members. Each commit is said to open a new *epoch* (group creation is at epoch 0), so the group secret at epoch n is more precisely called the *epoch secret* at n.

An epoch secret should only be known to the current members of the group in that epoch. Hence, the protocol seeks to ensure that members cannot read or write messages after they are removed, and new members cannot read old messages.

Secure Messaging. Within each epoch, the epoch secret is used to derive message encryption keys that members of the group can use to securely exchange application messages.

The protocol generates fresh encryption keys for each message to guarantee forward secrecy (FS): compromising a member's secrets should not allow the adversary to decrypt previous messages sent or received by that member. Note that the FS guarantee depends on the secure deletion of these messages on each member device [2, 33].

Furthermore, as long as each member regularly updates its encryption keys, the protocol provides post-compromise security: an adversary who learns a member's secrets at epoch n, but does not interfere until the member's keys are updated at epoch $m > n$, cannot decrypt messages after epoch m.

Decomposing MLS. Given this high-level view of MLS, the main cryptographic elements that need elaboration are: how a committer computes the new epoch secret and conveys it to the current group members, how application messages are encrypted in a way that provides authentication and forward secrecy, and how the protocol guarantees that all members of the group have a consistent view of the group membership and structure. In the remainder of this section, we describe sub-protocols of MLS that implement each of these components.

2.1 TreeKEM: Establishing Epoch Secrets

To participate in an MLS group, each endpoint e must first upload a signed *key package* containing its credential (including a signature verification key), a public encryption key pk_e, and other protocol parameters (e.g. supported ciphersuites). So, when a group member decides to add e to a group, it can use pk_e to encrypt a *welcome* package for e containing enough information for e to initialize its state and join the group.

At epoch 0, the epoch secret is derived from a fresh random value. Thereafter, at each commit, the committer computes a *commit secret* and sends it to all the members of the new epoch. Each member then mixes the commit secret with the previous epoch secret (at $n-1$) to obtain the new epoch secret (at n). A naive approach would be for the committer to generate a fresh commit secret and encrypt it for each group member using their public keys. However, in a group of size n, this design requires n expensive public-key encryptions for each commit, which does not scale well to large groups.

TreeKEM defines a more efficient commit operation by structuring the group as a complete binary tree, as depicted in Figure 1. The non-empty leaves of the tree contain the key packages of the current group members (a, b, c, d, e). Each internal node (also called *parent node*) corresponds to a subgroup consisting of the members underneath that node, and is associated with a *node secret* (e.g. s_{abc}) that is known only to its members (a, b, c). Each node secret (s_{abc}) is used to derive a public encryption key for the corresponding subgroup

(pk_{abc}). The node secret at the root (s_{abcde}) is known to the full group and is used as the commit secret for the epoch.

The benefit of the binary tree data structure is that when a committer (say b) wishes to convey a new commit secret to a group of size n, it only needs to compute and convey a single message containing $\log(n)$ public-key encryptions. Essentially, the committer (b) generates a fresh secret s_b, uses it to derive a sequence of node secrets for the path from its leaf to the root ($s'_{ab}, s'_{abc}, s'_{abcde}$), and conveys each new node secret to the rest of the subgroup by encrypting it under the corresponding sibling node's public key (pka, pk_c, pk_{de}). Each recipient (e.g. c) decrypts the node secret for the smallest subgroup it shares with the committer (s_{abc}), and derives the sequence of node secrets up to the root (s'_{abcde}).

In general, a parent node can be *blank*, or it may have *unmerged leaves*, which means that it is associated with not one but a set of public keys that collectively covers the members below that node. Consequently, the cost of each commit can actually vary between $\log(n)$ and n. TreeKEM also optimizes for the case when one of the children of a parent node is an empty subtree, by skipping the computation of that node's secret and treating it as blank, with the same public key as its non-empty subtree. Such nodes (e.g. the parent of de in Figure 1) are called *filtered nodes*.

We have given only a simplified summary of TreeKEM. The full TreeKEM protocol has many other details that we elide here, since they are unimportant for this paper. Several prior works have formally analyzed TreeKEM and its variants and shown that it implements a security definition called Continuous Group Key Agreement (CGKA) [3,4]. Other work has analyzed the way epoch secrets are derived in TreeKEM [21].

TreeKEM Tree Invariants. For our purposes, the pertinent observation is that the security of TreeKEM crucially relies on a *tree secrecy invariant*: if an internal node is associated with a node secret, then it can only be known to the members underneath that node. Recall that each parent node secret is chosen during a commit by a member below that node, but it is then encrypted under one of its children's public keys. Consequently, the TreeKEM secrecy invariant relies on the integrity of the public keys in the tree, which can be expressed as a *tree integrity invariant*: if an internal node is associated with a public encryption key, then this public key was computed for (a subset of) the current members of the node's subgroup, by one of the members of the subgroup.

2.2 TreeDEM: Group Message Encryption

Given an epoch secret, the TreeDEM protocol derives message encryption keys for each member in the current group and specifies how they are used to send and receive application messages, handshake messages (containing TreeKEM proposals and commits), and Welcome messages for new members. We briefly describe how TreeDEM authenticates and encrypts application messages.

Each application message is serialized into a bitstring along with metadata indicating the group, epoch, and sender. This bitstring is then signed with the sender's signing key (to authenticate which member sent the message) and then MACed with a key derived from current epoch secret. The serialized message, its signature, and MAC are then encrypted using an authenticated encryption (AEAD) scheme using the sender's current message encryption key. The recipient performs the reverse set of operations to decrypt the message, verify the signature and MAC to ensure that the message was sent by a sender who is a member of the group.

After each message is sent or received, the sender's message encryption key is updated (or *ratcheted*) using a key derivation function to provide forward secrecy: compromising a group member after a key has been updated does not reveal prior keys or messages encrypted under those keys.

Each group member needs to keep track of the current encryption keys for all members and update these keys at every application message. In large groups, maintaining all these keys can be costly, especially if only a small minority of members send messages in a each epoch. Consequently, TreeDEM uses a tree-based message key derivation technique that lazily derives keys for each sender, to reduce the number of keys each member needs to maintain.

The security functionality provided by TreeDEM is sometimes called Forward-Secure Group Authenticated Encryption with Associated Data (FS-GAEAD) and has been analyzed in prior work [4]. For the purposes of this paper, the pertinent feature of TreeDEM is its reliance on the group tree data structure for key derivation, and that it uses sender signatures to authenticate MLS messages.

2.3 TreeSync: Group State Synchronization

MLS relies on group members having a consistent view of the group state. Specifically all members must agree on (and authenticate) the membership of the group and the structure and contents of the public key tree (as depicted in Figure 1). Otherwise, a member may be fooled by an attacker into sending messages to groups it did not intend to communicate with.

Our key observation in this paper is that the task of synchronizing and authenticating the group state can be seen as an independent *generic* sub-protocol with minimal dependencies on TreeKEM and TreeDEM. This allows us to modularly analyze the group authentication guarantees of MLS without getting bogged down by the details of these other protocols.

We identify a protocol called TreeSync that encapsulates all operations on the MLS group state, while treating TreeKEM-related content as opaque bitstrings. We describe TreeSync in detail in the Section 3, and we formalize and analyze its integrity and authentication guarantees in Section 4, under minimal assumptions on TreeKEM and TreeDEM. We also show that TreeSync provides some of the prerequisites for the security of TreeKEM and TreeDEM, such as the TreeKEM

tree integrity invariant, and the Tree Hash construction.

Our treatment of TreeSync is in contrast to the MLS specification [8], which tightly interleaves its description of group synchronization with the key derivations of TreeKEM. Prior work on the authentication mechanisms in MLS [5] also follows this pattern by combining them with TreeKEM, which in our opinion results in unnecessarily complex proofs.

Authentication Attacks on MLS. Furthermore, many prior attacks on MLS can actually be better understood as attacks on TreeSync. For example, a *double join* attack occurs when a malicious member at leaf i manages to modify the content of a parent node that is not its ancestor [9]. In the *welcome message* attack, an attacker fools a new member into accepting a tampered tree with compromised public keys [14]. In the *tree signing attack*, the attacker changes the position of leaves in the tree to fool a new member [5]. Each of these attacks resulted in major changes to the protocol, significantly raising its complexity and reducing its efficiency. By identifying and analyzing TreeSync, we provide a formal framework for finding such attacks and evaluating defenses against them. Indeed, we identified flaws in the authentication and integrity mechanisms of MLS and fixed them during this work.

3 A Formal Specification of TreeSync

In this section, we describe the TreeSync protocol and its detailed formal specification in the F* language [43]. Unlike prior analyses of MLS that are based on high-level models written as pseudocode [3–5, 21], our F* specification is executable, and hence testable against the RFC test vectors and other MLS implementations. It accounts for all the low-level details of MLS, and so serves as both a formal companion to the RFC and a reference implementation.

The precision of our specification also means that our analysis is less likely to miss attacks. For example, in Section 4.5, we show a new attack that results from the ambiguity of the message formats between TreeSync and TreeDEM, and would not appear in more abstract models.

The authentication mechanisms of TreeSync are complex, with performance optimizations interleaved with cryptographic constructions. Our goal is to informally explain the design and its motivations, and guide the reader to the F* specification for full details. Our full specification and all proofs are included in supplementary material [1].

3.1 TreeSync data structures

We present a *generic* tree data structure that can be instantiated to obtain the TreeSync and TreeKEM trees. This is in contrast with the RFC, which combines the two trees.

```
type tree (leaf_t:Type) (node_t:Type) (l:nat) (i:tree_index l) =
  | TLeaf:
      data: leaf_t{l == 0} →
      tree leaf_t node_t l i
  | TNode:
      data: node_t{l > 0} →
      left: tree leaf_t node_t (l−1) (left_index i) →
      right: tree leaf_t node_t (l−1) (right_index i) →
      tree leaf_t node_t l i
```

The type tree left_t node_t l i describes a complete binary tree indexed by its height l – we follow the RFC convention that a standalone leaf has height 0. The type tree is parametric over leaf_t, the payload of the leaves, and node_t, the payload of the non-leaf (a.k.a. "parent") nodes. The leaves of the tree (i.e. the participants) are numbered left-to-right from 0 to $2^l - 1$. Hence, each leaf has an absolute *leaf index* that represents its position in the full tree.

We leverage F*'s dependent types to encode structural invariants on the tree. Notably, the i argument to tree enforces a correct-by-construction tracking of leaf indices, rules out programmer errors, and enforces the MLS invariant that two sub-trees at different positions, even if otherwise identical, are never interchangeable.

To obtain the TreeSync tree (called treesync), we instantiate the tree with the content of parent and leaf nodes:

```
type parent_node = {
  opaque_content: node_content;
  parent_hash: mls_bytes;
  unmerged_leaves: mls_list uint32; }

type leaf_node = {
  opaque_content: leaf_content;
  parent_hash: leaf_node_parent_hash;
  signature_key: signature_public_key;
  ...
  signature: mls_bytes; (* signs all the fields above, and more *) }

let treesync = tree (option leaf_node) (option parent_node)
```

The TreeSync tree must include some content provided by and useful for TreeKEM, such as public keys, but our specification treats these protocols as independent modules, as evidenced by the opaque_content fields: TreeSync is oblivious to the particular payload that its nodes carry.

Note that the mls_bytes type is a convenient abbreviation that enforces that the length of bytes we manipulate does not exceed $2^{30} - 1$, a requirement coming from the compact integer encoding of the QUIC standard [29], which MLS itself adopts. We refer the reader to the supplementary material for the full definitions of all our data structures.

Blank nodes and empty leaves. In the treesync datatype above, the leaf and parent node payloads are *optional*. Empty leaf nodes happen because the RFC mandates a complete binary tree, meaning some participant (leaf) nodes might be

empty, as illustrated in Figure 2. Empty parent (inner) nodes are called *blank nodes* in the RFC, and arise either from participant removals, or because of filtered nodes (§2.1).

3.2 TreeSync operations

TreeSync offers a series of group management operations that members can use to modify and synchronize the group state. In particular, any member can create a *proposal* message to suggest a change (e.g. add or remove a member) and send it to the rest of the group, via the Delivery Service (DS). A group member can then collect a set of proposals and send a *commit* message for these proposals along with a *path update*. None of these sending operations actually change the TreeSync tree; instead, each member waits for a commit to be accepted by the DS and sent back before executing the proposed changes. Hence, the DS resolves potential conflicts by choosing the order of commits for the whole group.

When a commit is processed, each of the proposals is executed in order to modify the local TreeSync state. In the rest of this subsection, we discuss how each of these changes is implemented. The key guiding principle for all the operations in TreeSync is that they must preserve the *tree integrity invariant*: every subtree with a non-blank node must have been authenticated by a participant at one of the leaves of the subtree. To enforce this invariant, TreeSync relies on the parent hash mechanism described in Section 3.4.

Processing Path Updates. Each commit operation ends with a *path update* that updates all the nodes on the path from the root down to the sender's leaf, updating the tree integrity mechanisms along the way. The function implementing path updates in F* has type as follows, where we omit boilerplate:

```
val apply_path: #l:nat → #li:leaf_index l 0 →
  t:treesync l 0 → p:pathsync l 0 li → treesync l 0
```

The apply_path function allows the client to update tree t of height l with a new path p, where p follows the path from root to leaf li, and carries fresh content for each node (including leaf) found along the path. The apply_path function, in addition to updating content along p, also updates the integrity protections at each node, as we will see later in Figure 3.

In line with our dependent type definition for trees, the leaf index li not only guarantees that the path terminates at participant (leaf) li, but also allows us to keep track of the leftmost leaf index as we move along the path. Once again, carrying such indices not only avoids errors, but greatly simplifies and automates our proofs. The 0 in the type signature is another invariant enforced "for free" by typing: this function is only intended to be called on a path starting at the root.

Processing Removal and Addition. The functions implementing add and remove have types as follows.

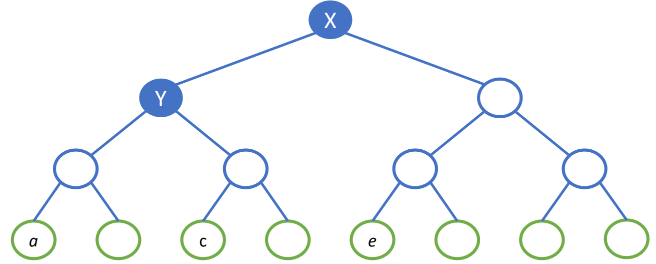

Figure 2: An MLS tree with three participants *a*, *c* and *e* at leaf indices 0, 2 and 4; other leaf nodes are empty. Due to the filtered node optimization, *X* and *Y* are the only parent nodes that are not blanked.

```
val tree_remove: #l:nat → #i:tree_index l → t:treesync l i →
  li:leaf_index l i → treesync l i
val tree_add: #l:nat → #i:tree_index l → t:treesync l i →
  li:leaf_index l i → ln:leaf_node → treesync l i
```

We note that adding a member may increase the size of the tree, and removal can shrink the tree. The full types of these functions include preconditions (omitted here, see [1]) that rule out various overflow conditions in various TreeSync structures whose length is bounded by the RFC.

If *a* wishes to remove *c* from the tree (Figure 2), MLS requires blanking out all of the nodes starting from *c* (a leaf), all the way up to, and including, the root. The net effect is that any cryptographic material that *c* may have authored is now gone from the tree. The RFC also mandates that each removal be enclosed in a commit that includes a path update by the committer *a*, which updates the contents of all nodes from *a* to the root, authenticates the removal, and restores integrity protections for the full tree at the root.

If a committed *e* wishes to add *b* to the tree (Figure 2), *e* fills out the first non-empty leaf (at index 1) with *b*'s data. The path update from *e* then updates the nodes between *e* and the root (e.g. *X*). However, there may be nodes between *b* and the root which are not updated (e.g. *Y*). These nodes will only be updated in a subsequent commit by one of the members under them (*a* or *b*). In the meantime, TreeSync performs supplemental book-keeping using *unmerged leaves*.

Unmerged leaves. Each node now contains a *list* of unmerged leaves (or *unmerged list*), with the invariant that participants in that list belong to the node's subtree. If *b* is in the unmerged list of *Y*, then it indicates that the addition of *b* to the subtree under *Y* postdates the modification of the node *Y*.

Addition now works as follows: after the first non-empty leaf has been filled with the new participant data, addition also extends the unmerged list of every node on the path from the new participant all the way up to the root. Concretely, after adding *b*'s data at leaf index 1, *e* inserts *b* into the unmerged list of *Y* and *X*. Only then does *e* issue a path update.

Path updates clear the unmerged list of each node they visit,

so when e issues a path update, the root's unmerged list is empty after the update, meaning that any integrity protections added to the root now cover the entire group – all is well.

Serialization and Parsing. Our specification implements the serialization and parsing of all the MLS data structures, for trees, messages, and signature contents down to the byte level – we follow the RFC to the letter. This is to be contrasted with all prior formal approaches, which study a *model* of MLS that is not guaranteed to be faithful to the RFC. As a consequence, our specification can actually be extracted and executed to establish that we are interoperable with test vectors and other implementations (§5); this level of precision also allowed us to find new attacks, such as signature collisions (§4.5).

3.3 Tree Hash

The MLS specification defines a tree hash operation that computes a digest for an entire tree; we rely on this operation in subsequent sections. The implementation details are of little importance for the rest of this paper: it suffices to say that the RFC implements an efficient recursive hash procedure akin to that of a Merkle Tree, and should two different MLS trees exhibit the same tree hash, then one has found a collision for the underlying hash function (§4.3). Proving this requires, naturally, reasoning about injectivity of serialization.

The tree hash provides an integrity mechanism for the MLS tree: if two members have the same tree hash they must have the same tree (barring hash collisions). Consequently, by including the tree hash within a signature, a sender can authenticate the full tree to a receiver. However, this integrity guarantee is not strong enough to protect new group members from tree tampering attacks by old members, such as the welcome message attack [14] and tree signing attack [5]. Consequently, MLS includes a second, stronger integrity mechanism called the Parent Hash.

3.4 Parent Hash

We have now exposed all of the TreeSync operations. Throughout our explanations, we have consistently referred to the need for a mechanism that can protect the integrity of the whole tree, while i) correctly accounting for both unmerged leaves and blank nodes mechanisms, and ii) satisfying the constraint that a path update can only modify nodes along the path. That integrity mechanism is known as the Parent Hash, and must accommodate further requirements: first, the number of hash computations and recomputations should be minimal (for efficiency); second, the parent hash should cover the contents of all the subtrees that existed in the tree when the parent hash was last modified.

Computation. Each node in the tree stores a parent hash. When a path update is applied, the parent hash of each node on the path is recomputed, starting at the root, and continuing

```
let rec apply_path_loop #l #i #li (t: treesync l i) (p: pathsync l i li)
  parent_parent_hash
= match t, p with
  // End of path: apply new contents from p onto t
  | TLeaf _, PLeaf lp → TLeaf (Some lp)
  | TNode _left right, PNode opt_opaque_content ps →
    let _, sibling = get_child_sibling t li in
    let opt_content' = // Compute the new node content
      match opt_opaque_content with
      | None → None // We skip filtered nodes
      | Some content → Some ({
        opaque_content = content;
        // Carried from previous loop iteration
        parent_hash = parent_parent_hash;
        // Notice we clear the unmerged leaves list
        unmerged_leaves = []; }) in
    // Compute the parent's parent hash for recursive call
    let parent_parent_hash' =
      match opt_opaque_content with
      | None → parent_parent_hash // We skip filtered nodes
      | Some content →
        compute_parent_hash content parent_parent_hash sibling
    in
    // Update the tree recursively
    if is_left_leaf li then ( // relative to tree of height l at position i
      let left' = apply_path_loop left ps parent_parent_hash' in
      TNode opt_content' left' right
    ) else (
      let right' = apply_path_loop right ps parent_parent_hash' in
      TNode opt_content' left right')
```

Figure 3: Implementation of the apply_path function, simplified. We write x' for the updated value for x.

all the way down to the leaf (participant) that issued the path update. We show the inner (recursive) loop of apply_path, in Figure 3.

To initialize recursion, the parent hash stored in the root node is always a special empty value. Then, at any given step along the way (with N the current node, S its sibling, and P their parent), the parent hash stored in N is updated to the hash of a serialized structure containing:

- the (new) parent hash stored in P,
- the (new) opaque payload stored in P, and
- the tree hash of S, which fully captures the contents of S, unmerged leaves included.

At the end of the path update issuance, the leaf signs its own parent hash. Doing so, the participant signs (authenticates) their own membership in the tree, as well as the content of their parent P, the entire sibling tree S, and whatever else the parent hash of P recursively covers. Transitively, this means that the leaf contains a hash value that protects the integrity of every node, sibling and parent, all the way up to the root.

Recall that when a node in the path has a blank subtree, it is called a *filtered node* and is treated as blank; in this case,

apply_path_loop skips the node and moves down to its child. Figure 2 illustrates this optimization: if c issues a path update, its parent node is *skipped*, and only Y and X get updated.

Each path update also clears the unmerged list of every node on the path, as the nodes that were in the unmerged list are now authenticated by the update. Accounting for unmerged leaves and filtered nodes significantly complicates the implementation of all the operations in TreeSync; this is one of the many reasons that motivate a formal proof.

Verification. Perhaps harder than updating the parent hash is *verifying* its correctness to prevent against malicious actors. This happens in two circumstances: first, upon joining a group; second, upon receiving a commit from another group member. In the first scenario, the whole tree must be visited; in the second scenario, this is only an incremental process wherein a lot of values from tree hash can be cached and reused.

Several subtleties arise in the process. We give an intuition for two of those, and leave a formal discussion of correctness properties to §4. First, a node N might have a non-empty unmerged list. This means that in order to validate the parent hash stored at N, one must consider the subtree at the time of the last authentication of N, that is, the subtree *without* the unmerged leaves. This requires introducing a new operation revert_add(P,leaves) operation that allows us to revert back the addition of a set of unmerged leaves (leaves) from a tree rooted at a parent node (P), so that we can compute a correct, albeit outdated, hash. The second complexity arises from the filtered nodes optimization. Notably, one must ensure that a malicious actor cannot surreptitiously introduce new nodes in an otherwise filtered (skipped) subtree.

Failing to account for both of these subtleties breaks our integrity invariant and can allow attacks on the protocol. Our authentication proof for TreeSync relies on a novel criterion, the "parent-hash link", that ties together the parent hash, the blank (skipped) nodes, and unmerged leaves together (§4).

This concludes our overview of the main elements of TreeSync, which itself only forms a small part of the MLS standard. The protocol is large enough and complex enough that we believe that it is hard, even for experts, to understand all the details, let alone reason about its security. We provide a testable specification for all of MLS in F* [1], which readers can inspect and run to hopefully gain a better understanding of the protocol and the machine-checked authentication theorems we proof for the TreeSync component.

4 A security proof of TreeSync

In this section, we describe a series of invariants and lemmas we prove for our TreeSync specification leading up to the main integrity and authentication guarantees of the protocol.

4.1 TreeSync State Invariants

As we saw in Section 3.1, the TreeSync tree data type already incorporates several structural invariants (complete tree, correct leaf index). In addition, we state and prove a series of invariants for all TreeSync states that are reachable by a sequence of operations. We describe three of these invariants, which play important roles in our security proofs:

Unmerged Leaves. At each parent node n, the unmerged leaves list must be sorted in increasing order, each unmerged leaf must point to a leaf index within the subtree rooted at n, and the leaf at this index must be non-blank.

This invariant can be easily checked for every TreeSync tree, and is necessary to prove the parent-hash invariant described below, but surprisingly, the latter two conditions were not required by the MLS draft. On our suggestion, they are now included since draft 15.

Leaf Validation. We require and enforce an invariant that all leaf signatures in the tree have been verified, and that the credential at each leaf has been issued (out-of-band) by the Authentication Service. Hence, we can assume that the verification key in the credential belongs to the member at the leaf and has been used to sign the leaf content. These are crucial pre-conditions for the authentication guarantees of TreeSync.

Parent-hash Linking. The parent hash construction (Section 3.4) creates links between parent nodes and their descendants. Formally, if a parent node P has two children C and S, we say that there is a *direct parent link* between C and P if, once we revert all the unmerged leaves of P (revert_add(P,P.unmerged_leaves)): (1) P and C are non-blank, (2) C has no unmerged leaves, and (3) C contains a parent-hash computed from P and S (C.parent_hash is equal to ParentHash(P.content, P.parent_hash, TreeHash(S))).

More generally, we say that there is a *parent link* from a descendant node D to P (written $D \rightsquigarrow P$) if P and D satisfy the conditions above and there is a path from D up to P such that all intermediate nodes on this path are filtered, i.e. they are blank and the corresponding sibling trees are fully blank. This generalization is needed because filtered paths may introduce blank nodes between a node and its linked parent.

We show that TreeSync enforces the invariant that each non-blank node P must have a descendant D such that $D \rightsquigarrow P$. By applying this invariant recursively, we obtain a more general notion of *path linking*: a leaf L is *path linked* to an ancestor node P, if all the non-blank nodes on this path $(T_1, \ldots, T_n)$ are sequentially parent linked ($T_i \rightsquigarrow T_{i+1}$). As we shall see, this is a crucial invariant for our authentication theorem.

F* Proofs. We formalize all our invariants on the TreeSync tree as a predicate which we attach to the treesync data structure as a *refinement type*. Thereafter, we use the F* type checker to prove that all TreeSync operations that modify the tree data structure preserve this predicate. The proofs rely

on some auxiliary lemmas but are mostly straightforward.

4.2 Verified Parsing and Serialization

Our F* specification includes parsers and serializers for all the byte formats defined in the MLS RFC, whether they represent trees, messages, or inputs to cryptographic constructions. We uniformly prove correctness properties for all these parsers and serializers, whether or not they belong in TreeSync.

In particular, for every MLS type T, we define a function serialize_T that translates T to bytes, and function parse_T that translates bytes to option T. We then prove that these functions are inverses of each other, and as a corollary, obtain that the serialization of each MLS type is *injective*.

```
∀(x:T). parse_T (serialize_T x) = Some x
∀(x:T) (b:bytes). parse_T b = Some x ⟹ serialize_T x = b
```

These properties are essential for functional correctness, but also for security. For example, the TreeHash construction relies on the serialization of a structure called TreeHashInput that includes the node type and hashes of the children (if any). We rely on the injectivity of this serialization to prove the integrity of TreeHash. Conversely, the failure of an injectivity lemma may point to an attack, as we will see in the case of the signature confusion attack on TreeSync authentication.

F* Proof. To prove all our parsers and serializers correct, we rely on a verified library of parser combinators in F* that largely automate the process of defining and verifying this code. This library allows us to write the RFC types as regular F* data types decorated with annotations describing how they should be serialized. Using F*'s *metaprogramming* feature, these types are automatically translated to parsers and serializers equipped with proofs of correctness.

4.3 Tree Hash Integrity Lemma

The TreeHash construction is used to verify the integrity of TreeSync trees: two members of a group can compare their tree hashes to verify if the trees are the same.

This integrity guarantee relies on the injectivity of TreeHash: if two subtrees t_1 and t_2 have the same tree hash (TreeHash(t_1) =TreeHash(t_2)), then either two trees are equal ($t_1 = t_2$), or else we can exhibit a pair of bitstrings b_1 and b_2 that exhibit a hash collision ($b_1 \neq b_2 \wedge H(b_1) = H(b_2)$).

In other words, a collision in TreeHash deterministically reduces to a collision in the underlying hash function. By structuring the lemma in this manner, we avoid making any symbolic or probabilistic assumption on hash functions.

The formal statement of this lemma in F* is given below:

```
val tree_hash_injectivity:
  #l1:nat → #i1:tree_index l1 → #l2:nat → #i2:tree_index l2 →
  t1:treesync l1 i1 → t2:treesync l2 i2 → Pure (bytes ∗ bytes)
  (requires tree_hash t1 == tree_hash t2)
  (ensures λ(b1, b2) →
    // Either the trees are equal and at the same position
    (l1 == l2 ∧ i1 == i2 ∧ t1 == t2) ∨
    // Or we computed a hash collision
    (hash b1 == hash b2 ∧ ¬(b1 == b2)))
```

Importantly, note that the lemma not only guarantees that the trees have the same content and structure, but also that they are at the same position, which is needed in the Parent Hash Integrity lemma below. The integrity of TreeHash is also relevant for TreeDEM, which authenticates the current tree hash in every message, hence guaranteeing that recipients and senders of each MLS message have the same tree.

F* Proof. Our proof of this lemma in F* is by induction on the structure of the two trees and case analysis on the TreeHash definition. It relies on the injectivity of serialization for the TreeHashInput type and as it travels down the trees, it inductively constructs the bitstrings that must exhibit the hash collision if the trees are not the same.

Our proof is similar to prior proofs for Merkle Trees (e.g. see [41, Section 7]). However, we note that even well known Merkle Tree implementations sometimes have subtle bugs [45], making them good targets for formal proof.

4.4 Parent Hash Integrity Lemma

Unlike the TreeHash, which is invalidated every time the tree is modified, the Parent Hash provides a more flexible integrity guarantee for subtrees that may, for example, have some unmerged leaves added after the last commit. To state the Parent Hash Integrity lemma, we first define a notion of tree equivalence that captures this flexibility, then define one step of the lemma before defining the lemma for the full tree.

Canonicalization and Equivalence. We define the canonicalization of a subtree T with respect to leaf index L, written canonicalize(T, L), by reverting the unmerged leaves at its root (revert_add(T, T.unmerged_leaves)) and by ignoring the signature value from leaf L. As we will see, if L is path-linked to T, canonicalize(T, L) captures precisely what is covered by L's signature. Because L's signature covers neither itself nor the unmerged leaves of T, we omit both in the canonicalization.

We say that two trees T and T' are equivalent with respect to a leaf index L, written $T \simeq_L T'$, if the two trees have the same canonicalization with respect to L.

Parent Link Integrity. Next we prove a lemma that shows how the parent link relation ($D \rightsquigarrow P$) protects the integrity of the tree. Consider two trees P_1 and P_2 where, P_1 has a descendant D_1 such that $D_1 \rightsquigarrow P_1$, and P_2 has a descendant D_2 such that $D_2 \rightsquigarrow P_2$. We prove that if $D_1 \simeq_L D_2$ then $P_1 \simeq_L P_2$.

That is, the parent link relation ($\rightsquigarrow$) enables us to lift the equivalence relation ($\simeq_L$) up the tree.

As with TreeHash, we state this lemma in terms of a function that either proves the equivalence of P_1 and P_2 or finds a hash collision. The statement of the lemma in F* is:

```
val parent_link_integrity:
  #ld1 → #ld2 → #lp1:nat{ld1 < lp1} → #lp2:nat{ld2 < lp2} →
  #id1:tree_index ld1 → #id2:tree_index ld2 →
  #ip1:tree_index lp1 → #ip2:tree_index lp2 →
  d1:treesync ld1 id1{node_has_parent_hash d1} →
  d2:treesync ld2 id2{node_has_parent_hash d2} →
  p1:treesync lp1 ip1{node_not_blank p1} →
  p2:treesync lp2 ip2{node_not_blank p2} →
  (* leaf index of L *) li:leaf_index ld1 id1 → Pure (bytes * bytes)
  (requires equivalent d1 d2 li ∧ parent_hash_linkedP d1 p1 ∧
     parent_hash_linkedP d2 p2) // Given the hypotheses
  (ensures λ(b1, b2) →
    equivalent p1 p2 li ∨ // Either the theorem is true
    (hash b1 == hash b2 ∧ ¬(b1 == b2))) // Or we have a collision
```

Parent Hash Integrity. By recursively applying the Parent Link Integrity lemma above, we obtain the full integrity guarantee for a path from a leaf to each of its ancestor nodes. Consider two trees T_1 and T_2, where T_1 has a leaf L_1 such that L_1 is path-linked to T_1, and T_1 has a leaf L_2 such that L_2 is path-linked to T_2. We show that if L_1 and L_2 have the same content and same leaf index, and if T_1 and T_2 have the same height, then $T_1 \simeq_L T_2$.

As a corollary, we obtain a lemma that is more directly useful for TreeSync authentication: if T_1 is the root node (i.e. its parent hash field is empty) , then T_2's height cannot be greater than T_1's, and all the subtrees between L_2 to T_2 must be point-wise equivalent to the corresponding subtrees on from L_1 to T_1. In practice, after every commit, the path update corresponds to a linked path from the committing leaf (e.g. L_1) to the root (T_1). However, as other leaves subsequently commit to the tree, the linked path no longer goes to the root and may be shorter (e.g. up to T_2).

F* Proofs. The proof for the parent link integrity lemma is similar to that of the Tree Hash integrity lemma. We rely on the injectivity of serialization, and the injectivity of tree hashes, and perform a case analysis on the parent hash definition to construct a hash collision if the two trees are not equivalent. The full parent hash integrity lemma is proved by induction on the length of the trees, propagating the hash collision up the tree. Due to the subtleties and many corner cases of the parent hash computation, we found that having a proof assistant like F* to check all the cases was quite valuable.

Weakness in Previous Drafts. We note that previous drafts of MLS (before draft 13) did not satisfy the Parent Hash Integrity lemma we state and prove in this section. This is because the parent hash construction did not include the Tree Hash of the sibling and instead only included the list of public keys (called the *resolution*) in the sibling tree, i.e. C.parent_hash is equal to ParentHash(P.content, P.parent_hash, Resolution(S)). Notably, the Resolution does not include the credentials of the leaves in S. This allows an adversary to tamper with the tree, by changing the leaf credentials in S, without it being detected via the parent hash mechanism.

Incidentally, the resolution mechanism was itself introduced in response to an attack (described in [5]) on the integrity protections of the parent hash mechanism in draft 9. Our analysis shows that there still are integrity attacks on the parent hash mechanism after this fix. We proposed the change to include the Tree Hash instead of the resolution and this was adopted in draft 13 of the standard. The change also has the benefit of more cleanly separating TreeSync mechanisms like parent hash from TreeKEM objects like public keys.

4.5 TreeSync Authentication Theorem

We can finally state the high-level TreeSync Authentication theorem. Consider the TreeSync tree T at group member b, obtained as a result of a valid sequence of TreeSync operations. Then, the theorem states that within every subtree T' of T where the root of the subtree is non-blank, there exists a leaf L in T' with a credential for some member a, such that either at some point in the past, the TreeSync tree at a contained the canonicalization of T' with respect to L (canonicalize(T', L)), or else a's signature key must have been compromised.

In other words, in every TreeSync state, every subtree with a non-blank root node is authenticated (up to the flexibility offered by equivalence) by one of the leaves in that tree. Notably, after a path update, the root of the full tree is guaranteed to neither be blank nor have unmerged leaves; the full TreeSync tree is thus always authenticated by some group member.

The authentication guarantee above is the first instance in our formal development where we are relating the state at one member (b) with the state at a different member (a). To formally state and prove this theorem, we need a runtime model that incorporates multiple parties and their interactions. To this end, we employ the DY* symbolic protocol framework.

Verifying Crypto Protocols with DY*. The DY* framework [15] defines a trace-based symbolic runtime model, where different *principals* can participate in protocols by calling cryptographic functions, generating keys and nonces, sending messages to each other, storing and modifying local state, and logging events to indicate authentication events. The attacker controls the network and can compromise principals: it can read and write any message, generate any number of keys, read the state of compromised principals, and store any amount of state for itself.

DY* implements a symbolic (or Dolev-Yao) abstraction of cryptographic functions, modeled using constructors and functions in F*. Here, we only use the hashing and signature functions in DY*. Hash functions are modeled as opaque one-way functions with no collisions. Signature schemes are

modeled as three functions: a key generation function that produces signature keypairs, a signature function that signs a bitstring using a signature key, and a verification function that takes a verification key, a bitstring, and its signature to verify. Verification succeeds if (and only if) the signature was computed with the signature function, meaning signatures are unforgeable unless the signature key is known to the attacker.

The trace-based runtime model and symbolic cryptographic assumptions of DY* are quite standard for symbolic verification and similar to models used in ProVerif [20] and Tamarin [35]. The main difference is the way proofs work in DY*. DY* is built as a library within the F* verification framework and hence has access to a rich higher-order dependently-typed programming language and a full-fledged theorem prover. Consequently, DY* is well suited to verify protocol implementations, and protocols with recursive data structures like trees, which automated provers like ProVerif and Tamarin struggle with. For example, DY* has been used to verify properties like PCS for recursive protocols like Signal [15] for an unbounded number of rounds. DY* has also been used to verify detailed protocol specifications like ACME [16] and protocol implementations like Noise* [28].

Applying DY* to TreeSync. The definitions we presented earlier (§3.1) are simplified ones. In reality, all of our TreeSync code is parametric over the type of bytes, and over operations on such bytes, which we achieve via F*'s *type class* mechanism. This allows us to write a single TreeSync, but instantiate it twice "for free": once with concrete bytes, to obtain an executable specification that can be tested over the wire, and once with symbolic DY* bytes. Similarly, our cryptographic primitives are either concrete, and call actual implementations; or symbolic, and annotated with DY* labels. To enable both concrete and symbolic crypto, we had to extend the DY* libraries with some missing features, like a proper treatment of bitstring lengths.

We then wrap the protocol code within a high-level API that offers functions for creating groups, adding and removing members, etc. This API internally stores session state for each open session, sends and receives messages, and logs events before each state change. This API is exposed to the attacker, so it can create any number of TreeSync sessions, and trigger any sequence of add, removes, and updates. However, the attacker does not get access to the internal state of uncompromised members. Our goal is to show that in all traces of honest TreeSync participants with the symbolic Dolev-Yao attacker, our confidentiality and authentication guarantees hold.

The first step is to typecheck that our protocol code obeys the DY* *labeling* discipline which ensures that secret values are kept secret; in TreeSync the only secrets are signature keys, which are used only to create signatures, so all data structures are labeled public, and the labeling proofs are straightforward.

Stating and Verifying TreeSync Authentication. Next, we need to annotate our code with *signature predicates* that describe all the possible uses of signatures in our full specification, including TreeKEM, TreeDEM, and TreeSync.

Within TreeSync, signatures are used only for leaf signatures. We require that before creating a leaf signature in a group g, the committer at leaf L must log an event of the form Send(g,canonicalize(T,L)), for every subtree T it modifies.

We can then state our authentication theorem as an invariant on the TreeSync state: in all reachable TreeSync session states at a member b of a group g, in every non-blank subtree T, there is a leaf L occupied by some principal a such that a previously logged an event of the form Send(g,canonicalize(T,L)), or else a was compromised. In DY*, this is stated as follows.

```
val treesync_authentication_theorem:
  #b:identity → #time:timestamp → #l:nat → #i:tree_index l →
  st:treesync_state → t:treesync l i →
  Lemma (requires
    is_reachable b time st ∧
    is_subtree_of t st.tree ∧
    root_node_is_not_blank t)
  (ensures ∃li, a. has_leaf_identity t li a ∧
    // a logged the corresponding Send event
    event_happened_before a time
      (Send st.group_id (canonicalize t author_li))
    // or was corrupted by the attacker before time
    ∨ is_corrupt a time)
```

To prove this theorem, we first rely on the unforgeability of signatures to show that the leaf signature in L ensures the existence of a linked path from L to T, and of corresponding Send events in the trace. We then combine the path-link invariant and the parent hash integrity lemma to conclude that the corresponding subtrees at b and a must be equivalent, and hence have the same canonicalization, to complete the proof.

Signature Confusion Attack. In fact, our first attempt at the authentication proof for TreeSync in draft 12 failed, because we were unable to prove that the attacker could not use a TreeDEM signature to forge a TreeSync signature. This is because both protocols use the same signature keys and there is an ambiguity between their signature formats. Consequently, we could not prove that the signature predicate for TreeSync is independent of the predicate used in TreeDEM.

This proof failure actually points to a real attack, and we can generate concrete instances of the signature contents used in the two protocols that collide after serialization. We note that this attack only appears if one models bitstring-level serialization (like our specification) since the two signatures would otherwise appear to be on different MLS types.

We presented this attack to the MLS working group and it was fixed as per our recommendation in draft 13. The fix uniformly disambiguates all signatures used in MLS for different purposes using different string labels. With this fix incorporated, we completed our authentication proof.

Interpreting TreeSync Authentication. The TreeSync authentication theorem tells us that the trees at different mem-

Component	F* LoC	Verification time
Library code	836	1min30s
TreeSync	1274	4min30s
TreeKEM	396	1min
TreeDEM	1384	2min45s
High level API	1024	1min30s
Library proofs	1170	1min45s
TreeSync proofs	4018	13min30s
Tests	2782	2min45s
Total specification	4914	11min15s
Total proofs	5188	15min15s

Table 1: Verification and coding effort for MLS on an Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz with 32GB of memory.

bers are consistent as long as enough honest (uncompromised) members keep creating commits. In particular, the theorem prevents all the known tree tampering attacks that plagued earlier versions of MLS [5, 14].

Interestingly, our proof makes no assumptions at all about TreeKEM, and our TreeSync specification treats all content provided by TreeKEM as opaque. We also do not make any assumptions about TreeDEM except for the signature disambiguation property described above. Consequently, TreeSync provides this authentication guarantee even if TreeKEM and TreeDEM were replaced by other (even broken) protocols.

Although we do not analyze TreeKEM and TreeDEM in this paper, TreeSync authentication is a necessary precondition for both these protocols, since they rely on tree agreement between members. We also prove that the authentication guarantee of TreeSync implies the TreeKEM tree integrity invariant, and that the tree hash used in TreeDEM provides strong integrity guarantees.

5 Implementation

MLS formal specification. Our complete F* specification totals 4914 lines of non-blank, non-comment code. We follow the modular approach described earlier (§2): our specification spans three namespaces, one for each subsystem. Table 1 gives a sense of how many lines of code (LoC) our implementation contains, grouped as run-time code, proofs, and tests.

Recall that we chose to materialize two trees for TreeSync and TreeKEM, favoring clarity and readability over conciseness; this tradeoff appears in numerous other places in our specification, where we always prefer a readable specification over a clever optimized implementation. For comparison, we evaluate mlspp and OpenMLS, two industrial implementations of MLS written in C++ and Rust respectively. The mlspp implementation, just like us, relies on an automated framework to derive parsers and serializers, and they use modern C++ with copious amounts of type inference to keep boilerplate to a minimum, totaling 4250 lines of non-blank, non-comment code. The OpenMLS implementation, in Rust, totals 15,000 lines of non-test, non-blank, non-comment code.

Measurement	This paper	mlspp	OpenMLS
Adds	2.7s	1.2s	0.7s
Messages	3.2s	0.6s	0.2s
Removes	5.5s	0.9s	0.7s

Table 2: Performance comparison between this paper and two other implementations of MLS. The time measured is the cumulative computation time for all participants in the group, measured on an Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz with 32GB of memory. *Adds:* Add 10 participants, one by one, with 20 messages from each participant after each add; *Messages:* Add 3 participants, with 400 messagers from each participant after each add; *Removes:* Add 15 participants, with 1 message from each participant after each add, then remove every participant with an odd position in the tree, then add participants until there are 15 participants again, with 1 message from each participant after each add.

Based on those two points of comparison, we conclude that we successfully managed to write a compact, concise, readable modular specification that can serve as a blueprint for any future MLS implementations.

MLS reference implementation. As mentioned earlier, our specification also serves as a reference implementation: all of our code is also fully executable. To run our code, we rely on F*'s extraction feature to produce OCaml code, which we then compile and execute using the standard OCaml toolchain. Our code interoperates with mlspp and OpenMLS and we participate in the IETF MLS interoperability meetings.

Our code requires numerous cryptographic primitives: we rely on the HACL* library [38, 41, 47] for those, thereby preserving the property that the entire codebase is verified. Furthermore, HACL* is one of the few libraries that support the latest version of HPKE, which we require for interoperability.

Performance evaluation. We compare our OCaml-extracted code to both mlspp (written in C++) and OpenMLS (written in Rust). We benchmark high-level integration tests that call the API functions for participant addition, participant removal, and sending of messages. The results are in Table 2.

We are comparing implementations written using different languages and toolchains. As such, we can only draw a broad conclusion, namely, that all implementations exhibit comparable performance, and generally execute within the same order of magnitude. We remark that our implementation, in spite of being written with no performance concerns in mind, still performs competitively. This means our code can be used off the shelf for rapid prototyping, interoperability testing, or generally, as a drop-in verified component when the highest degree of assurance is desired. Rudimentary profiling analysis indicates that a majority of the execution time is spent within the cryptographic primitives, which partially explains why

our implementation has only limited overhead.

We have several plans in the works to address the performance overhead. In the short term, we will investigate the use of better data structures (e.g. semi-persistent arrays) to make our pure, persistent byte manipulations more efficient. In the long run, we want to perform a proof of refinement that an efficient implementation, written in Rust, satisfies our high-level specification.

Skype integration. As a proof-of-concept, we integrated our reference implementation in a prototype version of the Skype messaging client. This was done as a one-time collaboration with a Microsoft team, where we added support in an experimental branch for a new feature called "secure group chats", powered by our MLS reference implementation. We tested and benchmarked the code on small groups exchanging a handful of messages. Overall, this allowed us to show that our code is deployable within a mainstream messenger.

Skype already features 1:1 private conversations using Signal; our implementation extended this functionality to actual groups. Skype is written using the Electron framework, i.e. a Web-based runtime environment. We used js_of_ocaml [46] to compile our extracted code to JavaScript, and linked it against HACL-WASM [39], a version of HACL* compiled directly to WebAssembly [27] while preserving security properties. The Skype team generously enabled the backend changes to implement the so-called Directory Service and Authentication Service that MLS relies upon.

We were able to successfully converse across endpoints, and there were no noticeable slowdowns in the user interface once we linked our code against efficient WASM-based cryptographic primitives. We conclude that the efficiency of MLS is bounded by the underlying cryptographic primitives, and that our reference implementation is a valid choice for security-conscious consumers.

6 Impact

Improving the standard. Our work identified several issues and attacks in the MLS drafts, and led to our proposing numerous changes that were ultimately adopted by the IETF.

The first issue we found was the signature confusion attack described in §4.5. We fixed this defect by uniformly adding labels to all signatures in MLS, to disambiguate their intent. This change was adopted in draft 13 and is required for our authentication theorem.

A second issue we found was that the integrity guarantee provided by the parent hash mechanism was too weak (§4.4), since it authenticated only TreeKEM related content in the tree. We proposed replacing this mechanism with one that uses the tree hash to authenticate the full content of the tree, including leaf credentials. This change, which enables our strong parent hash integrity lemma, was adopted in draft 13.

A third series of issues we found relates to the parent hash computation. In the process of performing the proof, we ended up with the four conditions for the well-formedness of the parent-hash link in the presence of unmerged leaves and filtered paths. We also identified several key criteria that must be met for the parent hash to recursively authenticate the whole tree, and for the corresponding inductive reasoning to succeed. The RFC was failing to enforce some of these, and we showed protocol traces that would break the TreeSync property.

Finally, we identified further well-formedness conditions for unmerged leaves that were not enforced upon joining a group (an unmerged leaf *must* point to a non-blank leaf). The protocol was missing this check, which we showed could break the parent-hash invariant. This is also fixed in draft 15.

Fixing Implementation Bugs. In addition to bugs in the standard itself, we also found implementation issues throughout the course of our interoperability testing. The first faulty implementation we identified was ours: we had some serialization errors, e.g. serializing a field as a `uint8` instead of `uint16`. We also found issues in both mlspp and OpenMLS, the two major industrial implementations of MLS at the time of writing. Both bugs were reported, and fixed.

A benefit of *executable* specifications is that they can be extensively tested for interoperability, like we did. This is the only way to gain confidence that the security theorem refers to the *actual* protocol, not a variant of it with an alternate serialization scheme. It is our opinion that any serious security analysis of a real-world protocol must include an executable specification; otherwise, one might prove properties over a different protocol, without realizing.

Lessons Learned. During our engagement with the IETF MLS standardization process, we found that the benefits of formal verification are now appreciated and understood when it comes to designing a new secure protocol. Notably, the MLS working group was highly reactive and appreciative of any bugs found by various teams; gladly accepts well-argued revisions and improvements; and, we posit, enjoys the added confidence that a formal analysis brings. We suspect that the many successes from the earlier TLS 1.3 have created a fruitful ground for this sort of collaboration.

Our approach of building an executable specification of the standard proved very useful for interactions with the working group. This not only makes security proofs much easier (as opposed to, say, having to perform them on a production codebase), but also allows rapid prototype and testing of proposed changes: for example, we were able to modify the specification and adapt the proofs to understand the security implications of a last-minute protocol modification. We encourage other standardization efforts to promote reference implementations written in high-level languages.

Conversely, MLS has grown to become a large protocol standard, and even understanding, let alone analyzing, the full protocol is a challenge even for cryptographic developers and

protocol experts. One of the contributions of this paper is the modular decomposition of MLS from a monolithic protocol into three independent components with a clean separation of concerns. In retrospect, this kind of modular design should have been built into the protocol standard itself, and perhaps should be a goal for the next version of MLS.

7 Related Work

Although group key establishment has been well studied in the literature (see e.g. [31,37]), group messaging differs from traditional group protocols in that it supports asynchronous messaging in dynamic groups. Unger et al. [44] provide a survey of messaging protocols, including some that support groups, conclude that "conversations between larger groups still lack a good solution". Since that survey, most academic work on group messaging has either been in the context of Signal or MLS. The extension of Signal with private authenticated groups was formally described and analyzed by Chase et al. [22], but Signal's sender-driven group messaging protocol does not scale to large groups. In the rest of this section, we compare our work with work on MLS and on other efforts to formally analyze cryptographic protocols.

Prior Analyses of MLS. The initial draft of MLS relied on Asynchronous Ratcheting Trees (ART) whose authors provide a cryptographic proof of their tree-based protocol design [23]. The original design of the TreeKEM protocol was presented in [13] and was adopted in MLS draft 2, and has since been extended with many features including blank nodes, unmerged leaves, and the proposal-commit pattern. Various versions of TreeKEM have been analyzed in a variety of security models. [3] presents a cryptographic analysis of TreeKEM in draft 7 against a passive, non-adaptative attacker, and defines continuous group key agreement (CGKA). They also analyze using Updatable Public Key Encryption to improve forward secrecy guarantees of TreeKEM. [4] modularly analyzes MLS in draft 11 against an active attacker. Their proof reason on high-level messages and miss the signature ambiguity attack, which we found by doing proofs on byte-level precise executable specifications. [21] analyzes the key derivation component of MLS in draft 11. [24] studies the multi-group security of MLS. All of these focus on the key exchange (TreeKEM) and data encapsulation (TreeDEM) components of MLS and do not consider tree integrity and authentication (TreeSync), our main focus.

Alwen et al. [5] study the security of TreeKEM against insider attacks and find a flaw on tree authentication. They propose different fixes by modifying the parent hash scheme, one of which is used in draft 16 and we study in this work (the "tree parent hash"). However, unlike this work, they study TreeKEM and TreeSync together as a monolithic protocol.

All the works mentioned above rely on manual pen-and-paper proofs. As the MLS standard grows, so do these manual proofs, making them hard to check and maintain. In this work, we use a formal verification tool to build a byte-level precise machine-checked specification for MLS that can be independently tested, modified, and verified.

A symbolic analysis of TreeKEM for forward security in Tamarin appears in [26] but it does not consider PCS or authentication. [14] uses F* to symbolically analyze TreeKEM in draft 7, finding an attack on tree authentication. However they do not identify TreeSync as an independent protocol and do not analyze the current parent hash design.

Mechanized Proofs of Crypto Protocols. Our approach follows a long line of work on the mechanized formal verification of cryptographic protocols (see [7] for a survey). Some protocol verification tools, like ProVerif [20], Tamarin [35], and DY* [15], rely on the *symbolic model* which treats cryptography abstractly and focuses on logical protocol flaws. Other tools, like CryptoVerif [19], EasyCrypt [11], and Squirrel [6], rely on the *computational model* which includes a more precise model of cryptography but provides less automation. Both kinds of tools have been applied to the analysis of real-world protocols like Signal [15,30] and TLS 1.3 [17,25].

Except for DY*, most existing tools struggle to analyze protocols with unbounded state (like trees) and with recursive structure (like ratcheting). Indeed, very little prior work applies to the mechanized analysis of group protocols [26,42] and even these works do not consider authenticated data structures like TreeSync trees.

Finally, many prior works verify reference implementations of protocols like Signal [40], Noise [28], and TLS [18]. Like us, these handle the full complexity of the protocol, including detailed message formats, yielding precise theorems that apply to running protocol code, not just abstract models.

8 Conclusion

We present a precise formal specification of the current version of the MLS protocol, along with a machine-checked proof of its TreeSync component. This work is part of a long-term engagement between the authors and the MLS working group, where we analyzed multiple intermediate versions of the protocol, found and fixed issues, and contributed design improvements to the protocol. Our specification consolidates our understanding of MLS and we hope it can serve as a formal guide to readers interested in this protocol.

Our proofs are only for TreeSync and do not cover TreeDEM and TreeKEM, although we formally specify and account for the interaction between TreeSync and these. We leave the comprehensive composite security analysis of all three components of MLS for future work.

Acknowledgments

We are indebted to Franziskus Kiefer, Raphael Robert and Richard Barnes for proofreading a draft of this paper and providing precious feedback. We are grateful to Jaroslav Franek for setting up a hackathon that allowed us to try out our MLS implementation in the Skype client, along with team members Jakub Kermaschek, Jurav Blazek, Lukas Liska and Katerina Cizkova.

This work received funding from the French Government, managed by the ANR under grant agreements ANR-22-PECY-006 and ANR-19-P3IA-0001.

References

[1] TreeSync: Supplementary material, 2022. https://github.com/Inria-Prosecco/treesync.

[2] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in Large-Scale urban protests: the case of hong kong. In *USENIX Security Symposium*, pages 3363–3380. USENIX Association, August 2021.

[3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In *CRYPTO*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277. Springer, 2020.

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1463–1483, 2021.

[5] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Paper 2020/1327, 2020.

[6] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *IEEE Symposium on Security and Privacy (S&P)*, pages 537–554. IEEE, 2021.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy (S&P)*, pages 777–795, 2021.

[8] R. Barnes, J. Millican B. Beurdouche, R. Robert, E. Omara, and K. Cohn-Gordon. The messaging layer security protocol. IETF Internet Draft, September 2022. version 16.

[9] Richard Barnes. Remove without double-join (in TreeKEM), 2018. https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik.

[10] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A Wood. RFC 9180: Hybrid public key encryption. Technical report, Internet Research Task Force, 2022.

[11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO*, pages 71–90, 2011.

[12] B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, A. Kwon, and A.Duric. Messaging layer security architecture. IETF Internet Draft, September 2022. version 9.

[13] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large dynamic groups a protocol proposal for messaging layer security (MLS). Research report, Inria Paris, May 2018.

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

[15] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[16] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. An in-depth symbolic security analysis of the ACME standard. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 2601–2617, 2021.

[17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–502. IEEE, 2017.

[18] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy, (S&P)*, pages 445–459, 2013.

[19] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[20] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2535–2553. IEEE, 2022.

[22] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1445–1459, 2020.

[23] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1802–1819, 2018.

[24] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In *USENIX Security Symposium*, pages 1847–1864. USENIX Association, 2021.

[25] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1773–1788, 2017.

[26] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. Cryptology ePrint Archive, Paper 2022/1130, 2022. https://eprint.iacr.org/2022/1130.

[27] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.

[28] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *IEEE Symposium on Security and Privacy (S&P)*, pages 107–124, 2022.

[29] Jana Iyengar and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. RFC 9000, May 2021.

[30] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno 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)*, pages 435–450. IEEE, 2017.

[31] Mark Manulis. Security-focused survey on group key exchange protocols. Cryptology ePrint Archive, Paper 2006/395, 2006. https://eprint.iacr.org/2006/395.

[32] Moxie Marlinspike. Private group messaging, 2014. https://signal.org/blog/private-groups/.

[33] Moxie Marlinspike. Disappearing messages for Signal, 2016. https://signal.org/blog/disappearing-messages/.

[34] Moxie Marlinspike and Trevor Perrin. Signal protocol, 2016. https://signal.org/docs.

[35] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[36] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/.

[37] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA*, pages 148–176. Springer International Publishing, 2021.

[38] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. Haclxn: Verified generic SIMD crypto (for all your favourite platforms). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 899–918, 2020.

[39] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1256–1274. IEEE, 2019.

[40] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1256–1274. IEEE, 2019.

[41] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (S&P)*, pages 983–1002. IEEE, 2020.

[42] Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David Basin. Automated verification of group key agreement protocols. In *2014 IEEE Symposium on Security and Privacy (S&P)*, pages 179–194, 2014.

[43] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[44] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *IEEE Symposium on Security and Privacy (S&P)*, pages 232–249, 2015.

[45] Forrest Voight. CVE-2012-2459 (block merkle calculation exploit). https://bitcointalk.org/?topic=102395, August 2012.

[46] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

[47] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.