End-to-End Encrypted Group Chats with MLS: Design, Implementation and Verification

TODO: insert here an easy to understand yet impactful figure representing MLS (don’t forget to fill this in before the final presentation!)

Théophile Wallez, Inria Paris
Jonathan Protzenko, Microsoft Research
Benjamin Beurdouche, Inria Paris, Mozilla
Karthikeyan Bhargavan, Inria Paris, Cryspen
Disclaimer

This talk is the long version of the USENIX Security ’23 talk:

TreeSync:
Authenticated Group Management for Messaging Layer Security

https://www.usenix.org/conference/usenixsecurity23/presentation/wallez

Internet defense prize and distinguished paper award!
What is Messaging Layer Security (MLS)
Secure group messaging
Secure group messaging


The New York Times

Signal Downloads Are Way Up Since the Protests Began

Organizers and demonstrators say they feel safer communicating with end-to-end encryption.
Secure group messaging


The New York Times

*Signal Downloads Are Way Up Since the Protests Began*

Organizers and demonstrators say they feel safer communicating with end-to-end encryption.
Secure group messaging


Signal Downloads Are Way Up Since the Protests Began

Organizers and demonstrators say they feel safer communicating with end-to-end encryption.

Forward secrecy

secure

compromise

time
Secure group messaging


Signal Downloads Are Way Up Since the Protests Began

Organizers and demonstrators say they feel safer communicating with end-to-end encryption.

Forward secrecy

- secure
- compromise

Post-compromise security

- healing
- secure
- compromise

time
Secure group messaging


The New York Times

Signal Downloads Are Way Up Since the Protests Began

Organizers and demonstrators say they feel safer communicating with end-to-end encryption.

Forward secrecy

<table>
<thead>
<tr>
<th>secure</th>
<th>healing</th>
<th>secure</th>
</tr>
</thead>
</table>

Post-compromise security

Eve joins

Eve leaves

time
State of the art, before MLS
State of the art, before MLS

Signal channels! Slow for large $N$, e.g. $N \approx 1000$

RFC 9420

Design constraints:
Secure, efficient, asynchronous, dynamic groups
State of the art, before MLS

N devices

$O(N^2)$ Signal channels!

Slow for large $N$, e.g. $N \approx 1000$
State of the art, before MLS

$N$ devices
$O(N^2)$ Signal channels!
Slow for large $N$, e.g. $N \approx 1000$

Design constraints:
Secure, efficient, asynchronous, dynamic groups

RFC 9420
A complex problem
A complex problem

https://nebuchadnezzar-megolm.github.io/

Upgrade now to address E2EE vulnerabilities in matrix-js-sdk, matrix-ios-sdk and matrix-android-sdk2

A complex problem

Upgrade now to address E2EE vulnerabilities in matrix-js-sdk, matrix-ios-sdk and matrix-android-sdk2


Many performance / security tradeoffs

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Create</th>
<th>Add</th>
<th>Remove</th>
<th>Update</th>
<th>Group Agreement</th>
<th>Update PPCS</th>
<th>Remove PACS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
<td>Recv</td>
<td></td>
</tr>
<tr>
<td>Sender Keys [18]</td>
<td>$N^2$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>Chained mKEM+</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$N$</td>
<td>$N$</td>
<td>Yes</td>
</tr>
<tr>
<td>2-KEM Trees+</td>
<td>$N$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>Yes</td>
</tr>
<tr>
<td>ART [7]</td>
<td>$N$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>TreeKEM+</td>
<td>$N$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>1</td>
<td>$\log(N)$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>TreeKEMB+</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\log(N)\ldots N$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>TreeKEMB+S+</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\log(N)\ldots N$</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Protocol: Protocol  
Performance: Performance  
Security: Security
A complex RFC

Table of Contents

1. Introduction
2. Terminology
   2.1. Presentation Language
   2.1.1. Optional Value
   2.1.2. Variable-Size Vector Length Headers
3. Protocol Overview
   3.1. Cryptographic State and Evolution
   3.2. Example Protocol Execution
   3.3. External Joins
   3.4. Relationships between Epochs
4. Ratchet Tree Concepts
   4.1. Ratchet Tree Terminology
   4.1.1. Ratchet Tree Nodes
   4.1.2. Paths through a Ratchet Tree
   4.2. Views of a Ratchet Tree
5. Cryptographic Objects
   5.1. Cipher Suites
   5.1.1. Public Keys
   5.1.2. Signing
   5.1.3. Public Key Encryption
   5.2. Hash-Based Identifiers
   5.3. Credentials
   5.3.1. Credential Validation
   5.3.2. Credential Expiry and Revocation
   5.3.3. Uniquely Identifying Clients
6. Message Framing
   6.1. Content Authentication
   6.2. Encoding and Decoding a Public Message
   6.3. Encoding and Decoding a Private Message
   6.3.1. Content Encryption
   6.3.2. Sender Data Encryption
7. Ratchet Tree Operations
   7.1. Parent Node Contents
   7.2. Leaf Node Contents
   7.3. Leaf Node Validation
   7.4. Ratchet Tree Evolution
   7.5. Synchronizing Views of the Tree
   7.6. Update Paths
   7.7. Adding and Removing Leaves
   7.8. Tree Hashes
   7.9. Parent Hashes
   7.9.1. Using Parent Hashes
   7.9.2. Verifying Parent Hashes
8. Key Schedule
   8.1. Group Context
   8.2. Transcript Hashes
   8.3. External Initialization
   8.4. Pre-Shared Keys
   8.5. Exporters
   8.6. Resumption PSK
   8.7. Epoch Authenticators
9. Secret Tree
   9.1. Encryption Keys
   9.2. Deletion Schedule
10. Key Packages
11. Group Creation
   11.1. Required Capabilities
   11.2. Reinitialization
   11.3. Subgroup Branching
12. Group Evolution
   12.1. Proposals
   12.1.1. Add
   12.1.2. Update
   12.1.3. Remove
   12.1.4. PreSharedKey
   12.1.5. ReInit
   12.1.6. ExtalnInit
   12.1.7. GroupContextExtensions
   12.1.8. External Proposals
   12.2. Proposal List Validation
   12.3. Applying a Proposal List
   12.4. Commit
   12.4.1. Creating a Commit
   12.4.2. Processing a Commit
   12.4.3. Adding Members to the Group
13. Extensibility
   13.1. Additional Cipher Suites
   13.2. Proposals
   13.3. Credential Extensibility
   13.4. Extensions
   13.5. GREASE
14. Sequencing of State Changes
15. Application Messages
   15.1. Padding
   15.2. Restrictions
   15.3. Delayed and Reordered Application Messages
16. Security Considerations
   16.1. Transport Security
   16.2. Confidentiality of Group Secrets
   16.3. Confidentiality of Sender Data
   16.4. Confidentiality of Group Metadata
   16.4.1. GroupID, Epoch, and Message Frequency
   16.4.2. Group Extensions
   16.4.3. Group Membership
   16.5. Authentication
   16.6. Forward Secrecy and Post Compromise Security
   16.7. Uniqueness of Ratchet Tree Key Pairs
   16.8. Keypacket Reuse
   16.9. Delivery Service Compromise
   16.10. Authentication Service Compromise
   16.11. Additional Policy Enforcement
   16.12. Group Fragmentation by Malicious Insiders
17. IANA Considerations
   17.1. MLS Cipher Suites
   17.2. MLS Wire Formats
   17.3. MLS Extension Types
   17.4. MLS Proposal Types
   17.5. MLS Credential Types
   17.6. MLS Signature Labels
   17.7. MLS Public Key Encryption Labels
   17.8. MLS Exporter Labels
   17.9. MLS Designated Export Pool
   17.10. The "message/mls" Media Type
18. References
   18.1. Normative References
   18.2. Informative References
Appendix A. Protocol Origins of Example Trees
Appendix B. Evolution of Parent Hashes
Appendix C. Array-Based Trees
Appendix D. Link-Based Trees
Contributors
Authors' Addresses

1,233 commits

0 Open ✓ 582 Closed
Quick interlude: our contributions
Contributions TL;DR
Contributions TL;DR

TreeSync  TreeKEM  TreeDEM
Contributions TL;DR
Contributions TL;DR
Contribution: Methodology

MLS

↓

F* specification
Contribution: Methodology

MLS

\[ \downarrow \]

F* specification

\[ \downarrow \]

Functional correctness proofs

(e.g. invariants)
Contribution: Methodology

- Contributions:
  - Symbolic implementation
  - Security proofs (for TreeSync)
  - Functional correctness proofs (e.g. invariants)

Diagram:
- MLS → DY
  - F* specification
  - Security proofs (for TreeSync)
Contribution: Methodology

![Diagram](image)

- **F* specification**
  - Symbolic implementation
  - Security proofs (for TreeSync)
  - Functional correctness proofs (e.g. invariants)

- **DY* to MLS**

- **HACL* to MLS**
  - Concrete implementation
  - Interoperability tests (4 implementations)
Contribution: Methodology

- Symbolic implementation
  - Security proofs (for TreeSync)
  - Functional correctness proofs (e.g. invariants)
  - Security proofs
  - Fix attacks

- Concrete implementation
  - Interoperability tests (4 implementations)

- Fix attacks
- DY* → MLS
  - F* specification
  - Functional correctness proofs
  - Security proofs
  - Concrete implementation
  - Interoperability tests (4 implementations)

- FIX
- HACL* → MLS

- Fix bugs
A tour of MLS
MLS decomposition

TreeSync: authenticated group synchronization
TreeKEM: efficient continuous group key establishment
TreeDEM: forward secure group messaging
The following explanations do the following assumption:
► there are $2^n$ participants in the group.

In particular, no dynamic groups (i.e. no add / remove).

Why:
► avoid consuming too much brainpower budget :)
► still give the core ideas behind MLS
TreeDEM
TreeDEM
TreeDEM

\[
\begin{align*}
H_a(r) & \quad H_b(r) & \quad H_c(r) & \quad H_d(r) \\
(a_0) & \quad (b_0) & \quad (c_0) & \quad (d_0) \\
H(a_0) & \quad H(b_0) & \quad H(c_0) & \quad H(d_0) \\
H(a_1) & \quad H(b_1) & \quad H(c_1) & \quad H(d_1) \\
H(a_2) & \quad H(b_2) & \quad H(c_2) & \quad H(d_2)
\end{align*}
\]
TreeDEM... with a tree

Hence:
Root key to participant key (amortized):
$O\left(1\right)$
TreeDEM... with a tree

\[ \text{Root key to participant key (worst case): } O(\log(n)) \]

But:

\[ \text{Root key to all participant keys (worst case): } O(n) \]

Hence:

\[ \text{Root key to participant key (amortized): } O(1) \]
TreeDEM... with a tree

\[
\begin{align*}
\text{Root key to participant key (worst case):} & \quad O(\log(n)) \\
\text{But:} & \quad O(n) \\
\text{Hence:} & \quad O(1)
\end{align*}
\]
TreeDEM... with a tree

Root key to participant key (worst case): $O(\log(n))$
TreeDEM... with a tree

Root key to participant key (worst case): $O(\log(n))$

But:
Root key to all participant keys (worst case): $O(n)$
TreeDEM...with a tree

Root key to participant key (worst case): $O(\log(n))$

But:
Root key to all participant keys (worst case): $O(n)$

Hence:
Root key to participant key (amortized): $O(1)$
TreeKEM, the initial idea (ART)

Invariant: private key of a node known exactly by its subtree.

$z = g^{xy}$
$x = g^{ab}$
$y = g^{cd}$
$a, g^a$
$b, g^b$
$c, g^c$
$d, g^d$

Send complexity: $O(\log(n))$ asymmetric operations
Receive complexity: $O(\log(n))$ asymmetric operations
TreeKEM, the initial idea (ART)

Invariant: private key of a node known exactly by its subtree.

\[
z' = g^{xy'}
\]

\[
x = g^{ab}
\]

\[
y' = g^{c'd}
\]

\[
a, \quad g^a
\]

\[
b, \quad g^b
\]

\[
c', \quad g^{c'}
\]

\[
d, \quad g^d
\]

Send complexity: \(O(\log(n))\) asymmetric operations
Receive complexity: \(O(\log(n))\) asymmetric operations
TreeKEM, the initial idea (ART)

Invariant: private key of a node known exactly by its subtree.

\[ z' = g^{xy'} \]

Send complexity: \( O(\log(n)) \) asymmetric operations
Receive complexity: \( O(\log(n)) \) asymmetric operations
TreeKEM, toward the final design

Idea: rely on asymmetric encryption (HPKE) and hashes (HKDF).
Invariant: private key of a node known exactly by its subtree.
Three steps: generate, encrypt, publish.
TreeKEM, toward the final design

Idea: rely on asymmetric encryption (HPKE) and hashes (HKDF).
Invariant: private key of a node known exactly by its subtree.
Three steps: **generate**, encrypt, publish.

\[
\begin{align*}
\text{pk}_x & \quad \text{sk}_x \\
\text{pk}_a & \quad \text{sk}_a \\
\text{pk}_b & \quad \text{sk}_b \\
\text{pk}_d & \quad \text{sk}_d
\end{align*}
\]

Send complexity: \( O(\log(n)) \) asymmetric operations
Receive complexity: only 1 asymmetric operation!
TreeKEM, toward the final design

Idea: rely on asymmetric encryption (HPKE) and hashes (HKDF).
Invariant: private key of a node known exactly by its subtree.
Three steps: generate, encrypt, publish.
TreeKEM, toward the final design

Idea: rely on asymmetric encryption (HPKE) and hashes (HKDF).
Invariant: private key of a node known exactly by its subtree.
Three steps: generate, encrypt, publish.
TreeKEM, toward the final design

Idea: rely on asymmetric encryption (HPKE) and hashes (HKDF). Invariant: private key of a node known exactly by its subtree. Three steps: generate, encrypt, publish.

Send complexity: $O(\log(n))$ asymmetric operations
Receive complexity: only 1 asymmetric operation!
TreeSync: why?

Alice joins a secure group, and receive a tree of public keys. How does she makes sure those keys are not attacker-controlled?
TreeSync: why?

Alice joins a secure group, and receive a tree of public keys. How does she makes sure those keys are not attacker-controlled?

How does she makes sure who is in the group? Can the attacker be in the group without her knowledge? Is Bob really Bob, or is it the attacker somehow?
TreeSync: why?

Alice joins a secure group, and receive a tree of public keys. How does she makes sure those keys are not attacker-controlled?

How does she makes sure who is in the group? Can the attacker be in the group without her knowledge? Is Bob really Bob, or is it the attacker somehow?

TreeSync solves these problems by authenticating TreeKEM’s state. In particular:

▶ authenticates all public keys, along with their recipients
▶ authenticates the roster, ensuring **group membership agreement**

Before the integration of TreeSync in MLS, several man-in-the-middle-like attacks were found in MLS. With TreeSync, this class of attacks are not possible anymore.
TreeSync: (naive) attempt 1

When a participant update keys, it signs the new tree.

$T_z =$

\[
\begin{align*}
&T_z = \text{sign}(T_z) \\
pk_z \\
pk_x \\
pk_a \\
pk_y \\
pk_b \\
pk_c \\
pk_d
\end{align*}
\]
TreeSync: (naive) attempt 1

When a participant update keys, it signs the new tree.

\[ T_{z'} = \]

Now, Alice’s signature is unintelligible! As a result, \( T_x \) not authenticated by Alice anymore.
TreeSync: (naive) attempt 1

When a participant update keys, it signs the new tree.

\[ T_{z'} = \]

\[ T_x = \]

Now, Alice’s signature is unintelligible!
As a result, \( T_x \) not authenticated by Alice anymore.
TreeSync: attempt 2

When a participant update keys, it signs the every modified subtree.

\[
\begin{align*}
& pk_z \\
& \quad pk_x \\
& \quad \quad pk_a \\
& \quad \quad \quad \text{sign}(T_a) \\
& \quad pk_b \\
& \quad \quad \text{sign}(T_x) \\
& \quad \quad \text{sign}(T_z) \\
& \quad pk_y \\
& \quad \quad pk_c \\
& \quad \quad \text{sign}(T_z) \\
& \quad \quad pk_d
\end{align*}
\]
TreeSync: attempt 2

When a participant updates keys, it signs every modified subtree.

\[ \text{sign}(T_a) \]
\[ \text{sign}(T_x) \]
\[ \text{sign}(T_z) \]

\[ \text{sign}(T_{c'}) \]
\[ \text{sign}(T_{y'}) \]
\[ \text{sign}(T_{z'}) \]
TreeSync: attempt 2

When a participant update keys, it signs the every modified subtree.

Invariant: every subtree is signed by one of the leaves under it.
Complexity: requires $\log(n)$ signatures in each leaf :(

Diagram:

```
pk_{z'}

  /     \
pk_x  pk_y'

  /   \
|    |
|    |
|    |
pk_a pk_b

  /   \
|    |
|    |
|    |
pk_{c'} pk_d

sign(T_a)  sign(T_x)  sign(T_z)

sign(T_{c'})  sign(T_{y'})  sign(T_{z'})
```
TreeSync: final attempt

\[
\begin{align*}
\text{sign}(pk_a, ph_a) \\
ph_a &= \text{hash}(pk_x, ph_x, T_B) \\
ph_x &= \text{hash}(pk_z, ph_z, T_Y)
\end{align*}
\]
TreeSync: final attempt

\[
\begin{align*}
sign(pk_x, ph_x) \\
ph_a &= \text{hash}(pk_x, ph_x, T_B) \\
ph_x &= \text{hash}(pk_z, ph_z, T_Y)
\end{align*}
\]
TreeSync: final attempt

$\text{sign}(\text{pk}_a, \text{ph}_a)$
$\text{ph}_a = \text{hash}(\text{pk}_x, \text{ph}_x, T_B)$
$\text{ph}_x = \text{hash}(\text{pk}_z, \text{ph}_z, T_Y)$
TreeSync: final attempt

$$\text{sign}(pk_{z'}, ph_{z'})$$

$$\text{hash}(pk_{y'}, ph_{y'}, T_D)$$

$$\text{hash}(pk_{z'}, ph_{z'}, T_X)$$

$$\text{hash}(pk_{x}, ph_{x}, T_B)$$

$$\text{hash}(pk_{z}, ph_{z}, T_Y)$$

$$\text{sign}(pk_{a}, ph_{a})$$

$$\text{sign}(pk_{c'}, ph_{c'})$$

Invariant: every subtree is linked by parent-hash to one of its leaves.
Complexity: requires only 1 signature in each leaf!
$2^n$ participants: what did we miss?

Blank leaves: for non-power-of-two number of participants

Blank nodes: remove participants and erase secrets they know

Unmerged leaves: add new participants efficiently

Filtered nodes: optimize away nodes that are redundant
Contributions on TreeSync
Contribution: Modularizing MLS

TreeSync: authenticated group synchronization
TreeKEM: efficient continuous group key establishment
TreeDEM: forward secure group messaging
Contribution: Modularizing MLS

TreeSync: authenticated group synchronization
TreeKEM: efficient continuous group key establishment
TreeDEM: forward secure group messaging
Contribution: Modularizing MLS

TreeSync: authenticated group synchronization
TreeKEM: efficient continuous group key establishment
TreeDEM: forward secure group messaging
def join_group(group):
    if well_formed(group):
        # ...
    else:
        raise MalformedGroupException

Desirable property: well_formed is an invariant under group modifications.
Contribution: Fixing TreeSync’s invariants

```python
def join_group(group):
    if well_formed(group):
        # ...
    else:
        raise MalformedGroupException
```

Desirable property: `well_formed` is an invariant under group modifications.

Actually, a well-formed group could become malformed!
Contribution: Fixing TreeSync’s invariants

```python
def join_group(group):
    if well_formed(group):
        # ...
    else:
        raise MalformedGroupException
```

Desirable property: `well_formed` is an invariant under group modifications.

Actually, a well-formed group could become malformed!
Contribution: Fixing TreeSync’s guarantees

7.9. Parent Hashes

While tree hashes summarize the state of a tree at point in time, **parent hashes** capture information about how keys in the tree were populated.

path. When a client computes an UpdatePath (as defined in Section 7.5), it computes and signs a **parent hash** that summarizes the state of the tree after the UpdatePath has been applied. These summaries are constructed in a chain from the root to the member's

As a result, the signature over the **parent hash** in each member's leaf effectively signs the subtree of the tree that hasn't been changed since that leaf was last changed in an UpdatePath. A new member joining the group uses these parent hashes to verify that the parent
Contribution: Fixing TreeSync’s guarantees

7.9. Parent Hashes

While tree hashes summarize the state of a tree at point in time, parent hashes capture information about how keys in the tree were populated.

path. When a client computes an UpdatePath (as defined in Section 7.5), it computes and signs a parent hash that summarizes the state of the tree after the UpdatePath has been applied. These summaries are constructed in a chain from the root to the member's

As a result, the signature over the parent hash in each member's leaf effectively signs the subtree of the tree that hasn't been changed since that leaf was last changed in an UpdatePath. A new member joining the group uses these parent hashes to verify that the parent

Problem 1: Guarantees described in imprecise prose.
Contribution: Fixing TreeSync’s guarantees

7.9. Parent Hashes

While tree hashes summarize the state of a tree at point in time, 
parent hashes capture information about how keys in the tree were 
populated.

As a result, the signature over the parent hash in each member's leaf 
effectively signs the subtree of the tree that hasn't been changed 
since that leaf was last changed in an UpdatePath. A new member 
joining the group uses these parent hashes to verify that the parent

Problem 1: Guarantees described in imprecise prose.

Problem 2: Guarantees not actually met by parent hash!
Contribution: Fixing TreeSync’s guarantees

7.9. Parent Hashes

While tree hashes summarize the state of a tree at point in time, parent hashes capture information about how keys in the tree were populated.

path. When a client computes an UpdatePath (as defined in Section 7.5), it computes and signs a parent hash that summarizes the state of the tree after the UpdatePath has been applied. These summaries are constructed in a chain from the root to the member's

As a result, the signature over the parent hash in each member's leaf effectively signs the subtree of the tree that hasn't been changed since that leaf was last changed in an UpdatePath. A new member joining the group uses these parent hashes to verify that the parent

Problem 1: Guarantees described in imprecise prose.

Problem 2: Guarantees not actually met by parent hash!
Contribution: Fixing a signature ambiguity attack

TreeSync

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_1}(\text{msg}_1)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_1}(\text{msg}_1)) \]

TreeDEM

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_2}(\text{msg}_2)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_2}(\text{msg}_2)) \]
Contribution: Fixing a signature ambiguity attack

TreeSync

\[
\text{sig} = \text{sign}(sk, \text{serialize}_{T_1}(msg_1)) \\
\text{verify}(pk, \text{sig}, \text{serialize}_{T_1}(msg_1))
\]

TreeDEM

\[
\text{sig} = \text{sign}(sk, \text{serialize}_{T_2}(msg_2)) \\
\text{verify}(pk, \text{sig}, \text{serialize}_{T_2}(msg_2))
\]

Same key
Contribution: Fixing a signature ambiguity attack

TreeSync

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_1}(\text{msg}_1)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_1}(\text{msg}_1)) \]

Same key

TreeDEM

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_2}(\text{msg}_2)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_2}(\text{msg}_2)) \]

Different types

What if \( \exists \text{msg}_1, \text{msg}_2, \text{serialize}_{T_1}(\text{msg}_1) = \text{serialize}_{T_2}(\text{msg}_2) \)?

Bad interaction between TreeSync and TreeDEM!

Attack found by doing proofs on a bit-precise specification, thanks to executability and interoperability tests.
Contribution: Fixing a signature ambiguity attack

TreeSync

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_1}(\text{msg}_1)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_1}(\text{msg}_1)) \]

TreeDEM

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_2}(\text{msg}_2)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_2}(\text{msg}_2)) \]

Same key

Different types

What if \( \exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T_1}(\text{msg}_1) = \text{serialize}_{T_2}(\text{msg}_2) \)?
Contribution: Fixing a signature ambiguity attack

What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T_1}(\text{msg}_1) = \text{serialize}_{T_2}(\text{msg}_2)$?
Bad interaction between TreeSync and TreeDEM!
Contribution: Fixing a signature ambiguity attack

TreeSync

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_1}(\text{msg}_1)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_1}(\text{msg}_1)) \]

TreeDEM

\[ \text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T_2}(\text{msg}_2)) \]
\[ \text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T_2}(\text{msg}_2)) \]

What if \( \exists \text{msg}_1, \text{msg}_2, \text{serialize}_{T_1}(\text{msg}_1) = \text{serialize}_{T_2}(\text{msg}_2) \)?
Bad interaction between TreeSync and TreeDEM!
Contribution: Fixing a signature ambiguity attack

What if ∃msg₁, msg₂, serialize_{T₁}(msg₁) = serialize_{T₂}(msg₂)?
Bad interaction between TreeSync and TreeDEM!

Attack found by doing proofs on a bit-precise specification, thanks to executability and interoperability tests.
Proof sketch of TreeSync
Security proof, step 1: invariants

We prove many invariants on TreeSync (the well-formedness checks):

- Leaf signatures are valid
- Every node is linked by parent-hash to a node under it
- Things with unmerged leaves
Security proof, step 2: the parent-hash guarantee theorem

We define an equivalence relation on trees $\simeq$.

We prove the theorem:

\[
\begin{array}{c}
P_1 \quad \simeq \\ \uparrow \\ C_1 \end{array}
\quad \quad \quad \quad \quad
\begin{array}{c}
P_2 \\ \uparrow \\ C_2 \end{array}
\]
We define an equivalence relation on trees $\sim$.

We prove the theorem:

$$
P_1 \sim P_2
$$

$$
C_1 \sim C_2
$$
Security proof, step 3: signature invariant

We want to prove: every subtree is authenticated by one of its leaves.

Proof sketch:

\[
T_1 \quad T_2 \quad \ldots \quad T_n
\]

\[
T_1 \quad T_2 \quad \ldots \quad T_n \approx \approx \approx
\]
Security proof, step 3: signature invariant

We want to prove: every subtree is authenticated by one of its leaves.

Proof sketch:

\[ \begin{array}{c}
T_n \\
\uparrow \\
\vdots \\
\uparrow \\
T_2 \\
\uparrow \\
T_1 \\
\approx \\
T_1' \\
\uparrow \\
T_2' \\
\uparrow \\
T_n' \\
\end{array} \]
Security proof, step 3: signature invariant

We want to prove: every subtree is authenticated by one of its leaves.

Proof sketch:

\[
\begin{array}{c}
T_1 \\
T_1' \\
\vdots \\
T_2 \\
T_2' \\
T_n \\
T_n'
\end{array}
\]

\[\cong\]

\[\cong\]
Security proof, step 3: signature invariant

We want to prove: every subtree is authenticated by one of its leaves.

Proof sketch:

\[ T_n \rightleftharpoons T'_n \]
\[ \cdots \rightleftharpoons \cdots \]
\[ T_2 \rightleftharpoons T'_2 \]
\[ T_1 \rightleftharpoons T'_1 \]
Final notes
Proof effort

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

Roughly two man-years of work, because many by-products to work on:

- Develop the methodology to treat such large protocols
- How to obtain a bit-precise specification
- Developed a framework for verified message formatting, both concrete and symbolic (in submission at CCS!)
- A protocol during its standardization is a moving target
Conclusion

Our contributions:
- formally specify MLS decomposed into three sub-protocols: TreeSync, TreeKEM, and TreeDEM
- prove the security of TreeSync in the Dolev-Yao model
- do proofs on an executable, interoperable specification
- found design flaws and submitted fixes to the MLS Working Group

Future work: security proofs for TreeKEM and TreeDEM; prove efficient implementations.

The MLS Working Group gladly welcomed these contributions, resulting in a fruitful collaboration.

<> https://github.com/Inria-Prosecco/treesync
✉️ theophile.wallez@inria.fr
🌐 https://www.twal.org/
🐦 @twallez