TreeSync: Authenticated Group Management for Messaging Layer Security

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

TODO: insert here an easy to understand yet impactful figure representing MLS (don’t forget to fill this in before the final presentation!)
What is Messaging Layer Security (MLS)
Secure group messaging
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.
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

Signal Downloads Are Way Up Since the Protests Began

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

Forward secrecy

Post-compromise security

Secure

Healing

Secure

Eve joins

Eve leaves

State of the art, before MLS

+N devices

$O(N^2)$

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

\[ O(N^2) \] 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. $N \sim 1000$

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

(https://inria.hal.science/hal-02425229/)

<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>Sender Keys [18]</td>
<td>$N^2$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chained mKEM+</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</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>1</td>
</tr>
<tr>
<td>ART [7]</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>1</td>
</tr>
<tr>
<td>TreeKEM+</td>
<td>$N$</td>
<td>$\log(N)$</td>
<td>$\log(N)$</td>
<td>1</td>
<td>1</td>
<td>$\log(N)$</td>
<td>1</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>1</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>1</td>
</tr>
</tbody>
</table>

Protocol | Performance | Security

https://nebuchadnezzar-megolm.github.io/
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 Join
   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. ExternalInit
   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. KeyPackage 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
Our contributions
Contributions TL;DR
Contributions TL;DR

TreeSync

TreeKEM

TreeDEM
Contributions TL;DR

TreeSync  TreeKEM

TreeDEM
Contributions TL;DR
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
Contribution: Formal proof of TreeSync

MLS → F* specification
Contribution: Formal proof of TreeSync

MLS

$\Rightarrow$

F* specification

$\Rightarrow$

Functional correctness proofs

(e.g. invariants)
Contribution: Formal proof of TreeSync

MLS

\[ \text{F* specification} \]

\[ \text{Functional correctness proofs} \quad (\text{e.g. invariants}) \]

\[ \text{Symbolic implementation} \]

\[ \text{Security proofs} \quad (\text{for TreeSync}) \]

\[ \text{DY*} \]
Contribution: Formal proof of TreeSync

- Symbolic implementation
  - Security proofs (for TreeSync)

- Functional correctness proofs (e.g. invariants)

- Concrete implementation
  - Interoperability tests (4 implementations)

Diagram:

\[ 	ext{MLS} \rightarrow \text{F* specification} \rightarrow \text{DY*} \rightarrow \text{Symbolic implementation} \rightarrow \text{Security proofs (for TreeSync)} \]

\[ \rightarrow \text{HACL*} \rightarrow \text{Concrete implementation} \rightarrow \text{Interoperability tests (4 implementations)} \]
Contribution: Formal proof of TreeSync

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

Fix attacks

DY*

F* specification

Fix bugs

HACL*

Concrete implementation

Interoperability tests (4 implementations)
Contribution: Formal proof of TreeSync

Symbolic implementation

Security proofs (for TreeSync)

Fix attacks

Functional correctness proofs (e.g. invariants)

Concrete implementation

Interoperability tests (4 implementations)
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.
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. 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 path. 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

\[
\begin{align*}
\text{sig} &= \text{sign}(sk, \text{serialize}_{T_1}(msg_1)) \\
\text{verify}(pk, \text{sig}, \text{serialize}_{T_1}(msg_1))
\end{align*}
\]

TreeDEM

\[
\begin{align*}
\text{sig} &= \text{sign}(sk, \text{serialize}_{T_2}(msg_2)) \\
\text{verify}(pk, \text{sig}, \text{serialize}_{T_2}(msg_2))
\end{align*}
\]
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

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

\[
\text{verify}(pk, \text{sig}, \text{serialize}_{T_2}(msg_2))
\]
Contribution: Fixing a signature ambiguity attack

```
sig = \text{sign}(sk, \text{serialize}_T(msg))
verify(pk, sig, \text{serialize}_T(msg))
```

Same key

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

Different types

What if $\exists \, msg_1, msg_2, \text{serialize}_{T_1}(msg_1) = \text{serialize}_{T_2}(msg_2)$?

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

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

TreeDEM

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

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

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

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

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

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.
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