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

https://www.nytimes.com/2020/06/11/style/signal-messaging-app-encryption-protests.html

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

https://www.nytimes.com/2020/06/11/style/signal-messaging-app-encryption-protests.html

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.

time

Secure group messaging

https://www.nytimes.com/2020/06/11/style/signal-messaging-app-encryption-protests.html

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

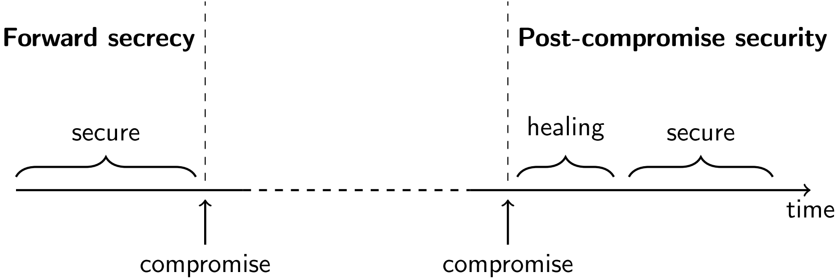
Secure group messaging

https://www.nytimes.com/2020/06/11/style/signal-messaging-app-encryption-protests.html

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.



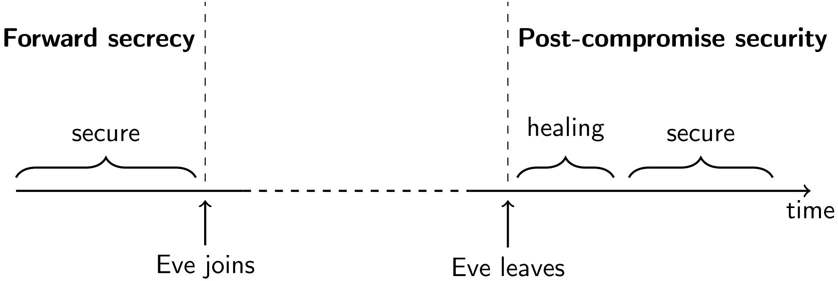
Secure group messaging

https://www.nytimes.com/2020/06/11/style/signal-messaging-app-encryption-protests.html

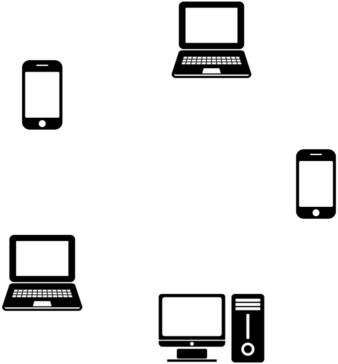

The New York Times

Signal Downloads Are Way Up Since the Protests Began

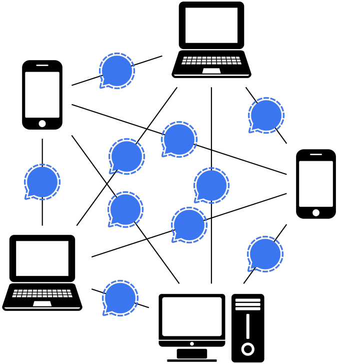
Organizers and demonstrators say they feel safer communicating with end-to-end encryption.



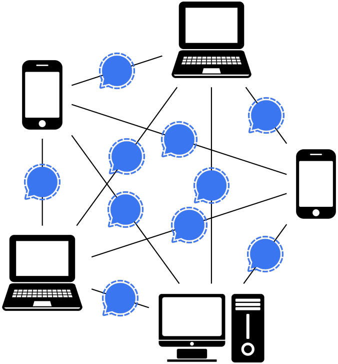
State of the art, before MLS



State of the art, before MLS

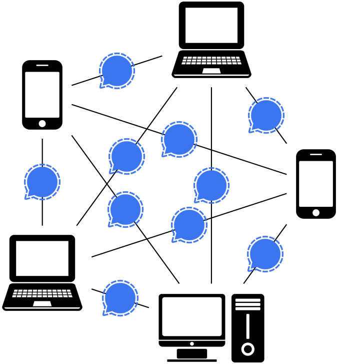


State of the art, before MLS



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

State of the art, before MLS



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

RFC 9420

Design constraints:
Secure, efficient, asynchronous, dynamic groups

A complex problem

A complex problem

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

[matrix]

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

28.09.2022 17:41 — Security — Matthew Hodgson, Denis Kasak, Matrix Cryptography Team, Matrix Security Team

A complex problem

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

[matrix]

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

28.09.2022 17:41 — Security — Matthew Hodgson, Denis Kasak, Matrix Cryptography Team, Matrix Security Team

Many performance / security tradeoffs

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

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

Protocol — Performance — 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
  10.1. KeyPackage Validation
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 Expert 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

[Page 164]

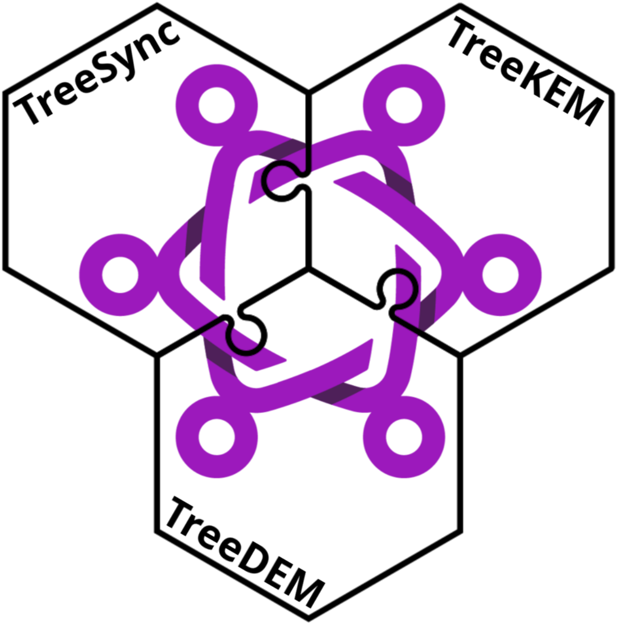
1,233 commits

0 Open ✓ 582 Closed

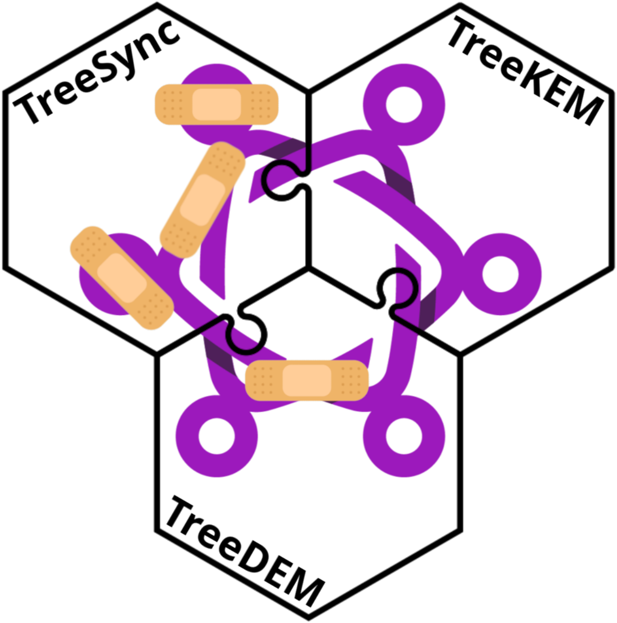
Quick interlude: our contributions

Contributions TL;DR

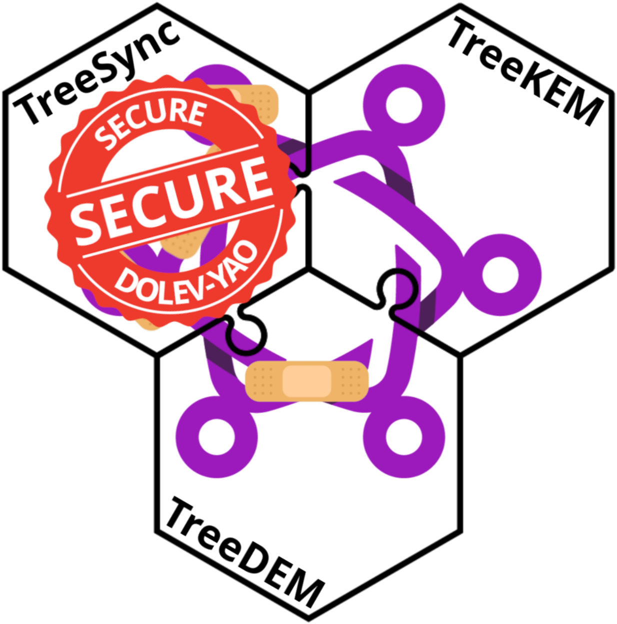
Contributions TL;DR



Contributions TL;DR



Contributions TL;DR



Contribution: Methodology

MLS

↓

F* specification

Contribution: Methodology

MLS

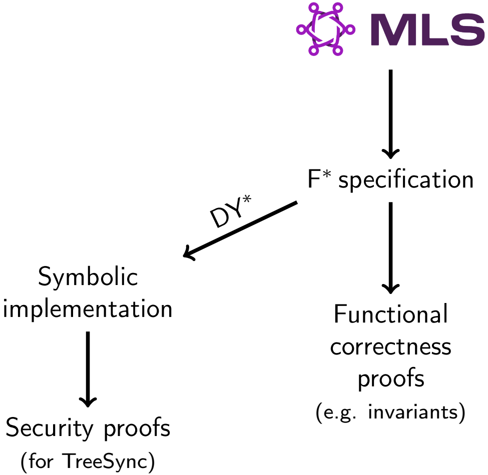
↓

F* specification

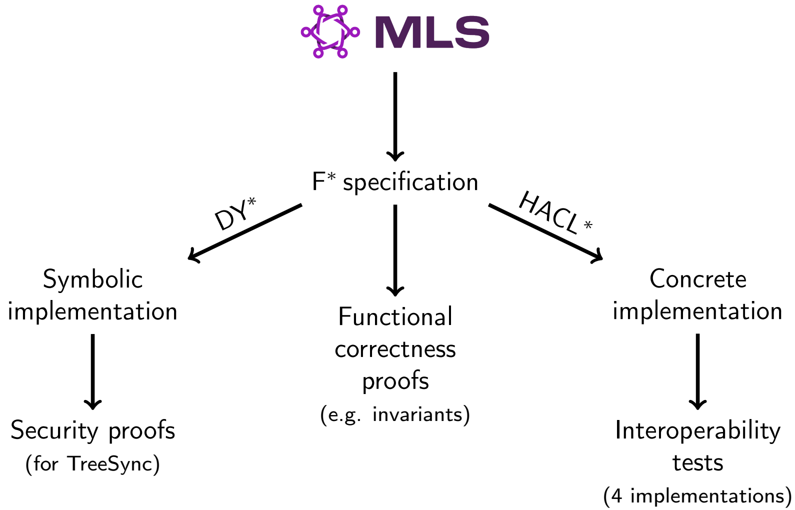
↓

Functional
correctness
proofs
(e.g. invariants)

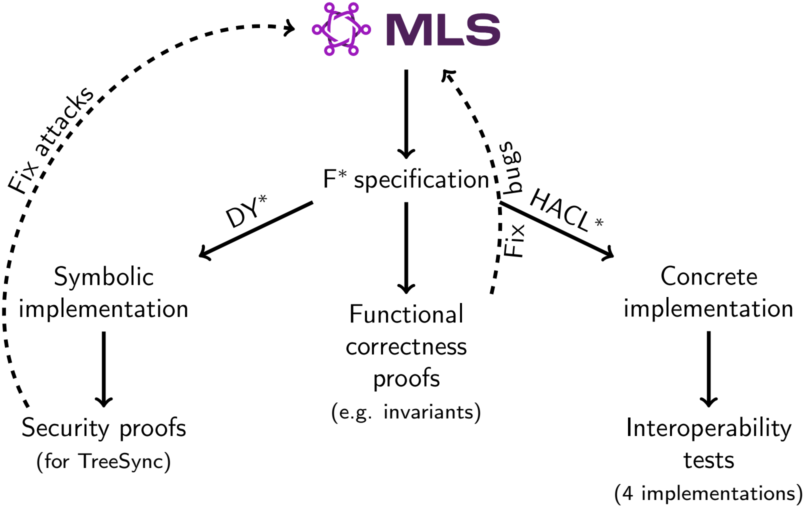
Contribution: Methodology



Contribution: Methodology

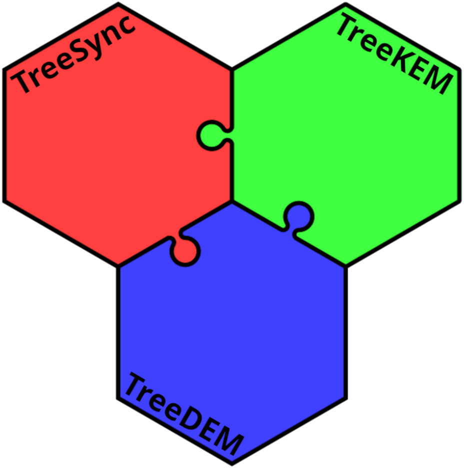


Contribution: Methodology



A tour of MLS

MLS decomposition



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

Disclaimer

The following explanations do the following assumption:

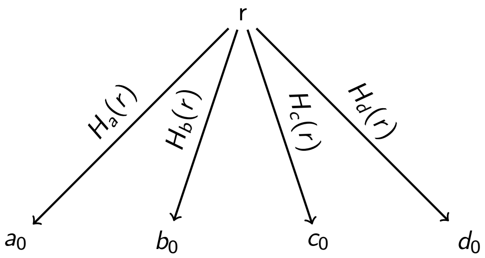
- there are 2^n participants in the group.

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

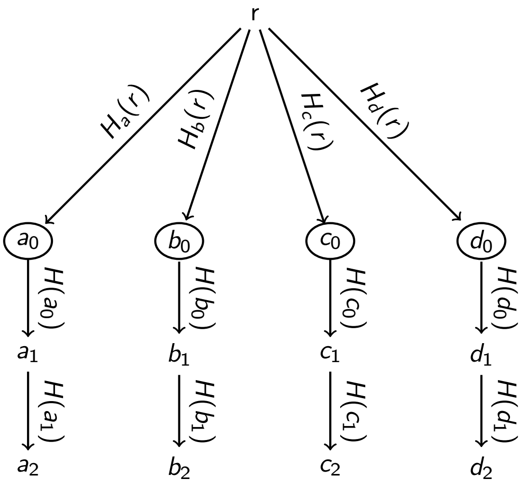
Why:

- avoid consuming too much brainpower budget :)
- still give the core ideas behind MLS

TreeDEM

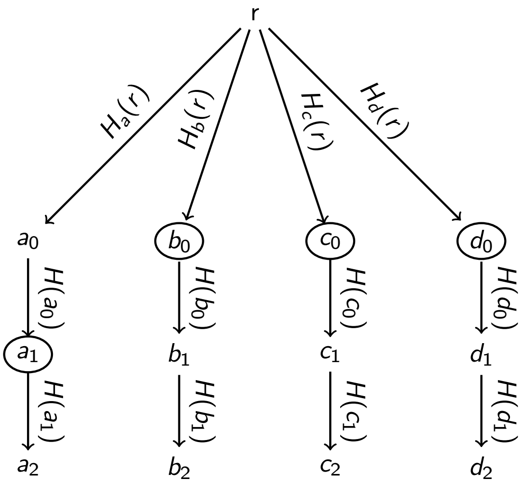


TreeDEM

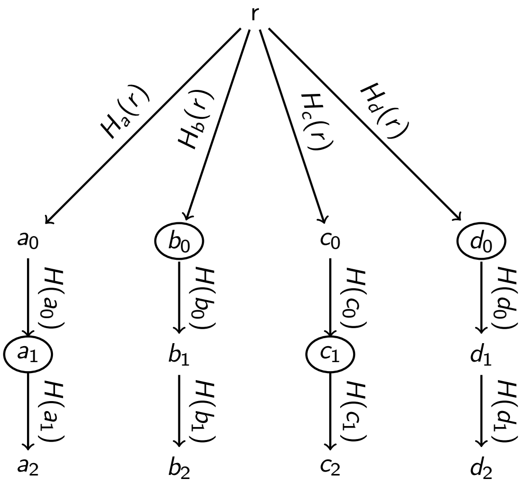


r
$H_a(r)$
$H_b(r)$
$H_c(r)$
$H_d(r)$
a_0
b_0
c_0
d_0
$H(a_0)$
$H(b_0)$
$H(c_0)$
$H(d_0)$
a_1
b_1
c_1
d_1
$H(a_1)$
$H(b_1)$
$H(c_1)$
$H(d_1)$
a_2
b_2
c_2
d_2

TreeDEM

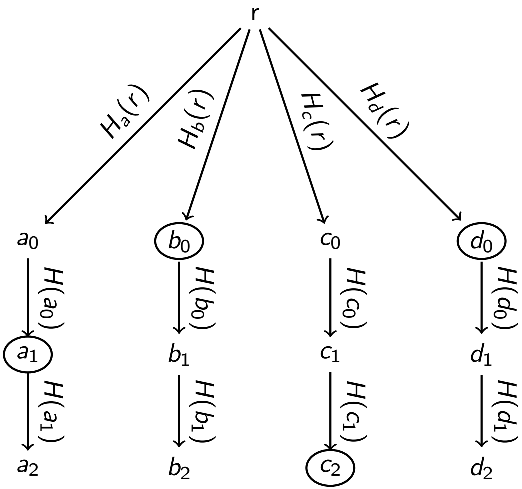

r
$H_a(r)$
$H_b(r)$
$H_c(r)$
$H_d(r)$
a_0
b_0
c_0
d_0
$H(a_0)$
$H(b_0)$
$H(c_0)$
$H(d_0)$
a_1
b_1
c_1
d_1
$H(a_1)$
$H(b_1)$
$H(c_1)$
$H(d_1)$
a_2
b_2
c_2
d_2

TreeDEM

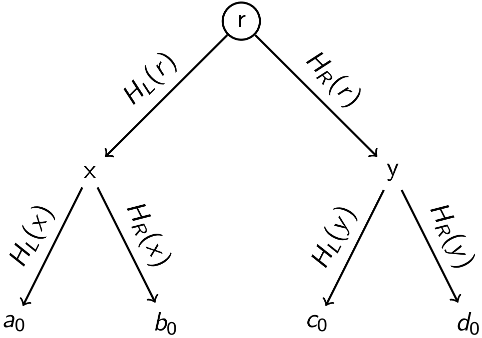


r
$H_a(r)$
$H_b(r)$
$H_c(r)$
$H_d(r)$
a_0
b_0
c_0
d_0
$H(a_0)$
$H(b_0)$
$H(c_0)$
$H(d_0)$
a_1
b_1
c_1
d_1
$H(a_1)$
$H(b_1)$
$H(c_1)$
$H(d_1)$
a_2
b_2
c_2
d_2

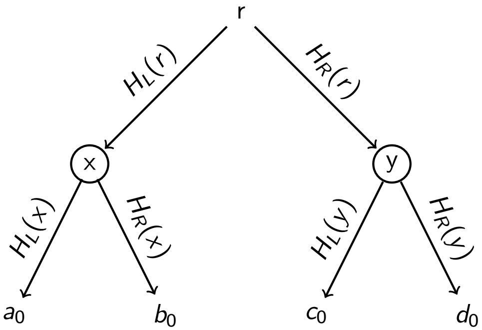
TreeDEM


r
$H_a(r)$
$H_b(r)$
$H_c(r)$
$H_d(r)$
a_0
b_0
c_0
d_0
$H(a_0)$
$H(b_0)$
$H(c_0)$
$H(d_0)$
a_1
b_1
c_1
d_1
$H(a_1)$
$H(b_1)$
$H(c_1)$
$H(d_1)$
a_2
b_2
c_2
d_2

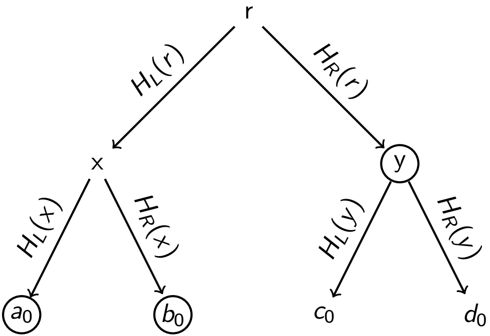
TreeDEM. . . with a tree



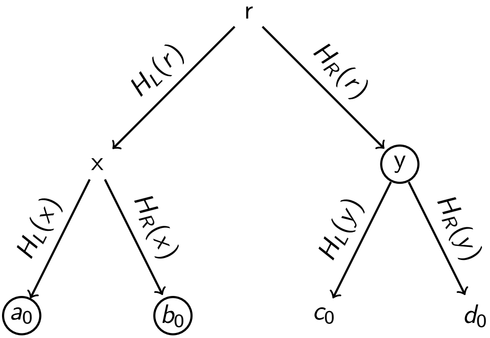
TreeDEM. . . with a tree



TreeDEM... with a tree

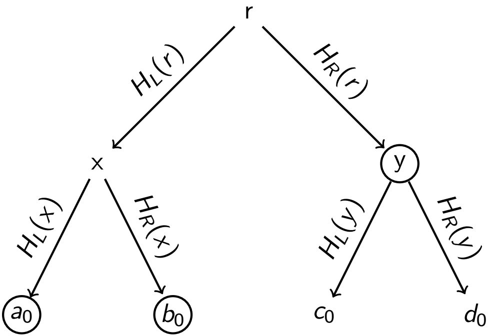


TreeDEM...with a tree



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

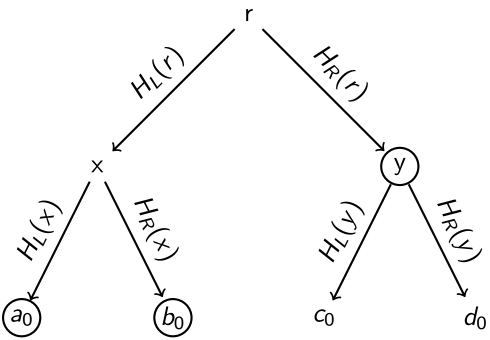
TreeDEM. . . with a tree



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



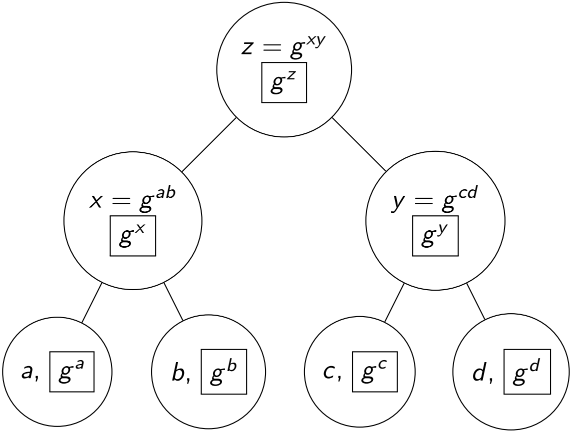
Root key to participant key (worst case): $O(\log(n))$

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)

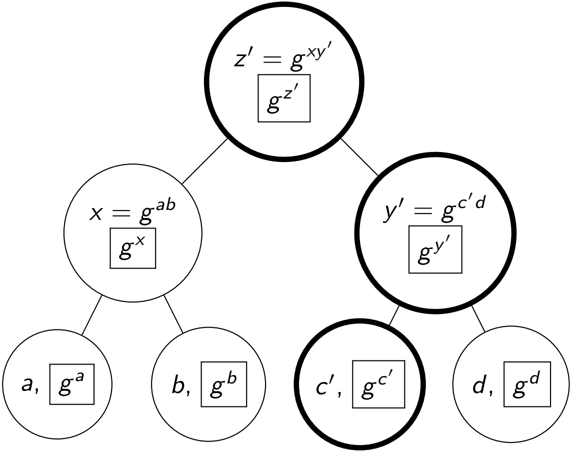
Idea: do a tree of Diffie-Hellman.
Invariant: private key of a node known exactly by its subtree.



TreeKEM, the initial idea (ART)

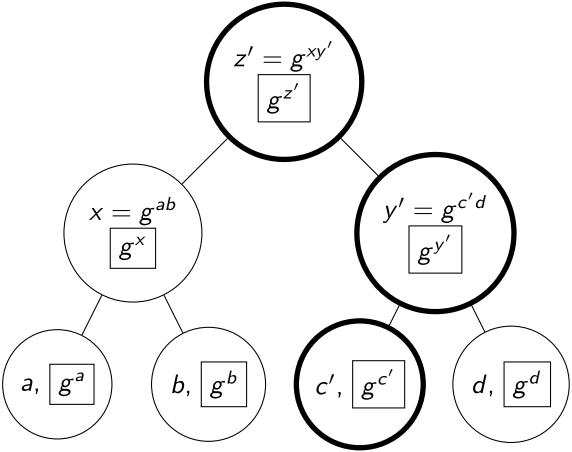
Idea: do a tree of Diffie-Hellman.

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



TreeKEM, the initial idea (ART)

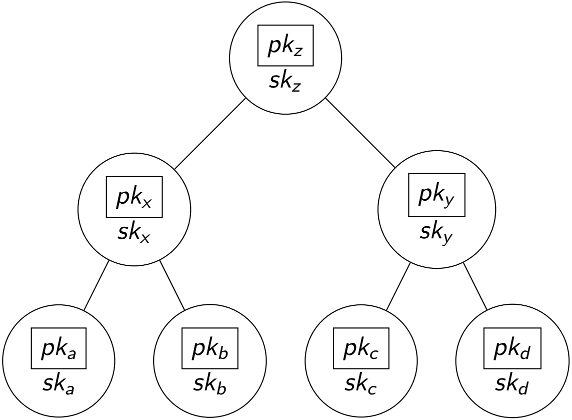
Idea: do a tree of Diffie-Hellman.
Invariant: private key of a node known exactly by its subtree.



Send complexity: $O(\log(n))$ asymetric operations
Receive complexity: $O(\log(n))$ asymetric operations

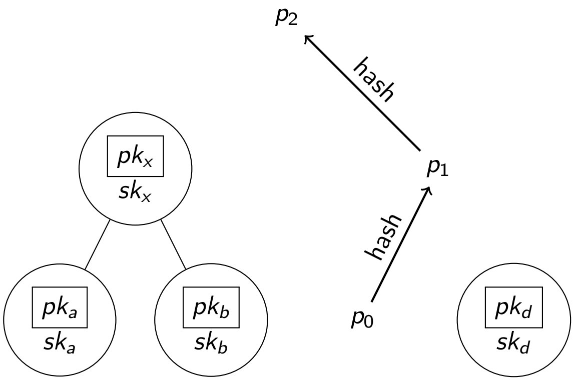
TreeKEM, toward the final design

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



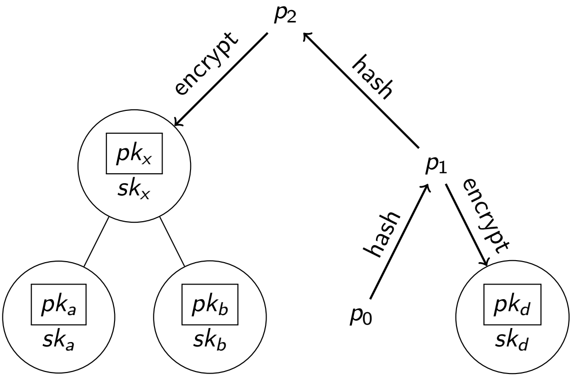
TreeKEM, toward the final design

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



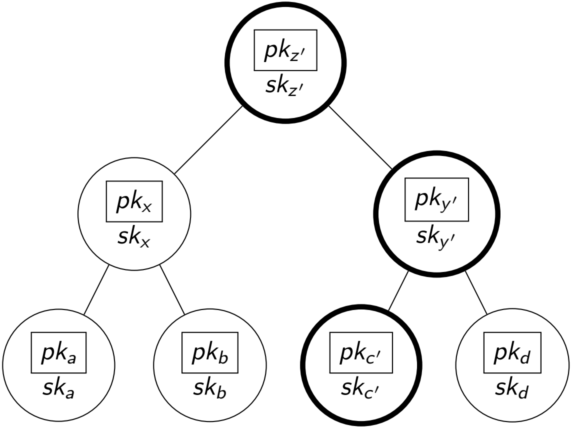
TreeKEM, toward the final design

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



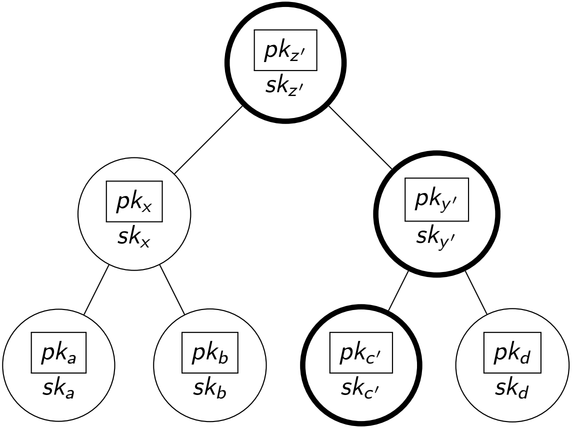
TreeKEM, toward the final design

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



Send complexity: $O(\log(n))$ asymetric operations
Receive complexity: only 1 asymetric 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?

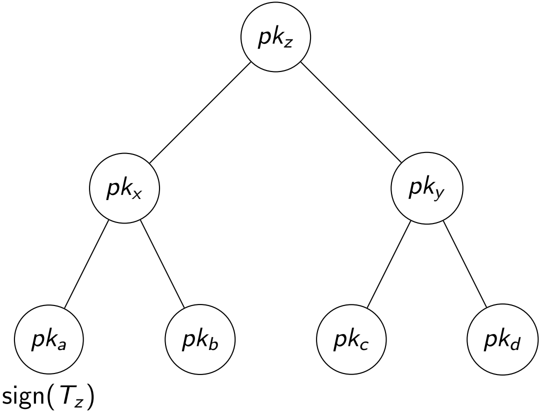
TreeSync solves these problems by authenticating TreeKEM's state.
In particular:

- 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.

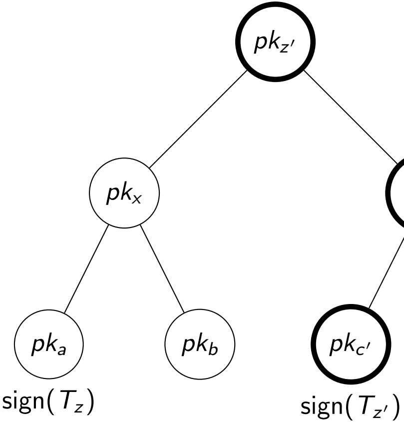
TreeSync: (naive) attempt 1

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



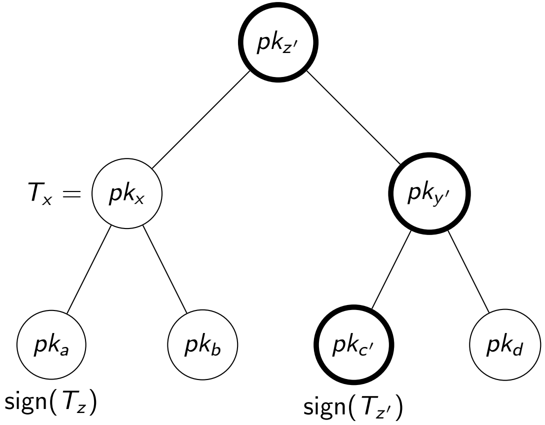
TreeSync: (naive) attempt 1

When a participant update keys, it signs the new tree.
$T_{z'} =$



TreeSync: (naive) attempt 1

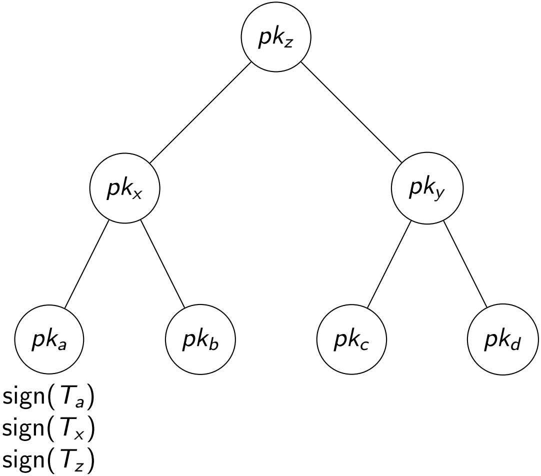
When a participant update keys, it signs the new tree.
$T_{z'} =$



Now, Alice's signature is unintelligible!
As a result, T_x not authenticated by Alice anymore.

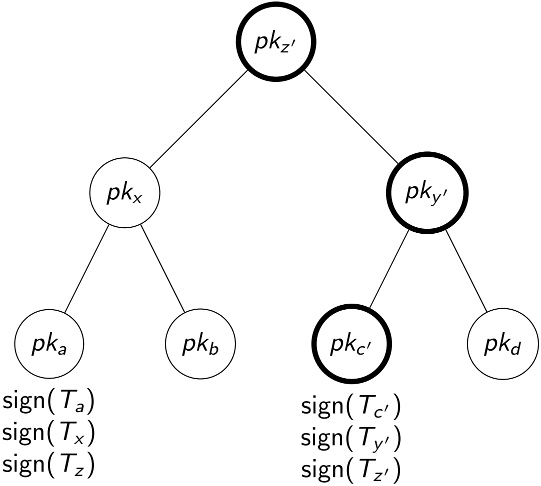
TreeSync: attempt 2

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



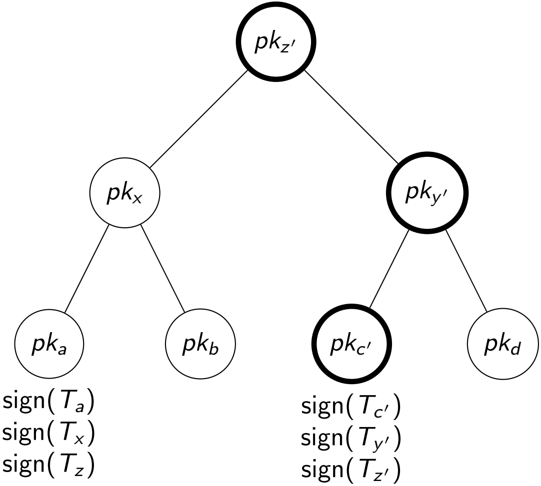
TreeSync: attempt 2

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



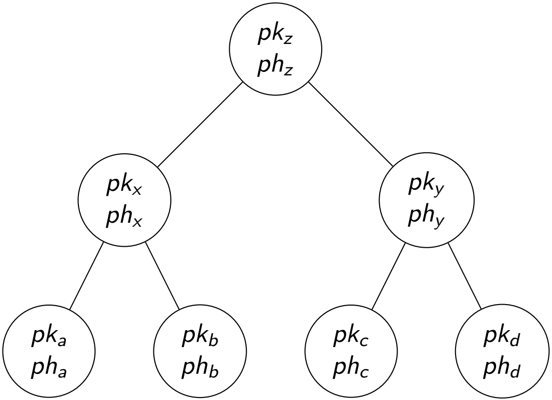
TreeSync: attempt 2

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



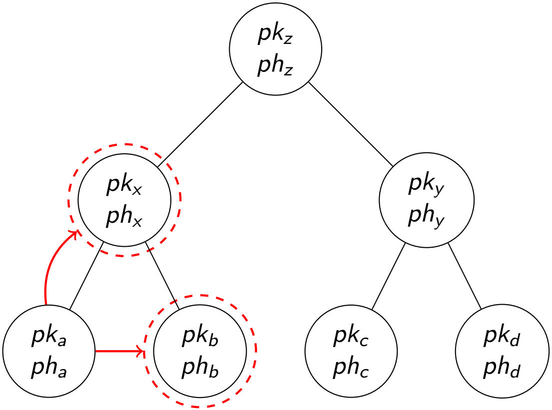
Invariant: every subtree is signed by one of the leaves under it.
Complexity: requires $\log(n)$ signatures in each leaf :(

TreeSync: final attempt



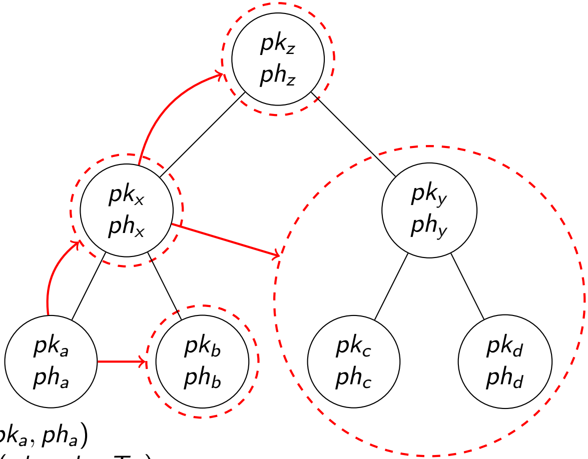
$$\text{sign}(pk_a, ph_a)$$
$$ph_a = hash(pk_x, ph_x, T_B)$$
$$ph_x = hash(pk_z, ph_z, T_Y)$$

TreeSync: final attempt



$$\text{sign}(pk_a, ph_a)$$
$$ph_a = hash(pk_x, ph_x, T_B)$$
$$ph_x = hash(pk_z, ph_z, T_Y)$$

TreeSync: final attempt

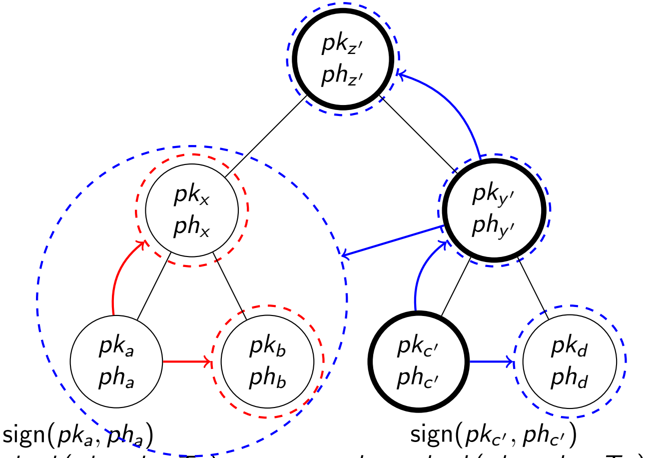


$\text{sign}(pk_a, ph_a)$

$ph_a = hash(pk_x, ph_x, T_B)$

$ph_x = hash(pk_z, ph_z, T_Y)$

TreeSync: final attempt



$$ph_a = hash(pk_x, ph_x, T_B) \qquad ph_{c'} = hash(pk_{y'}, ph_{y'}, T_D)$$

$$ph_x = hash(pk_z, ph_z, T_Y) \qquad ph_{y'} = hash(pk_{z'}, ph_{z'}, T_X)$$

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

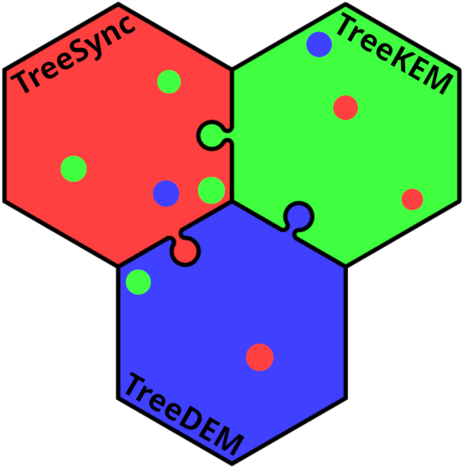
Blank nodes: remove participants and erase secrets they know

Unmerged leaves: add new participants efficiently

Filtered nodes: optimize away nodes that are redundant

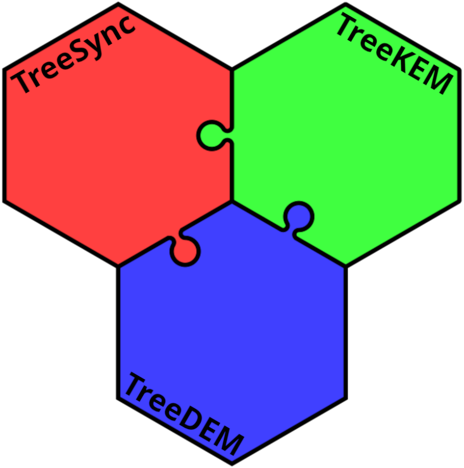
Contributions on TreeSync

Contribution: Modularizing MLS



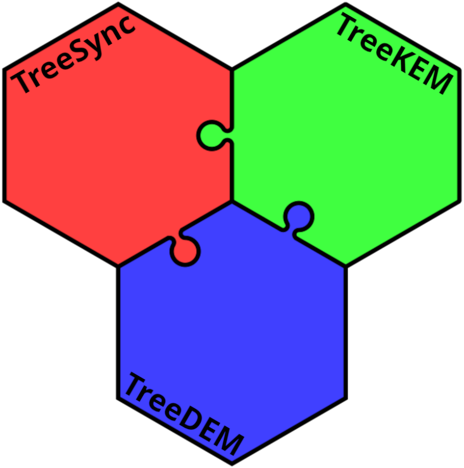
TreeSync: authenticated group synchronization
TreeKEM: efficient continuous group key establishment
TreeDEM: forward secure group messaging

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

Contribution: Fixing TreeSync's invariants

```
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

```
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

```
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.

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 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

$\text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T1}(\text{msg}_1))$

$\text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T1}(\text{msg}_1))$

TreeDEM

$\text{sig} = \text{sign}(\text{sk}, \text{serialize}_{T2}(\text{msg}_2))$

$\text{verify}(\text{pk}, \text{sig}, \text{serialize}_{T2}(\text{msg}_2))$

Contribution: Fixing a signature ambiguity attack

Same key

TreeSync

$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$

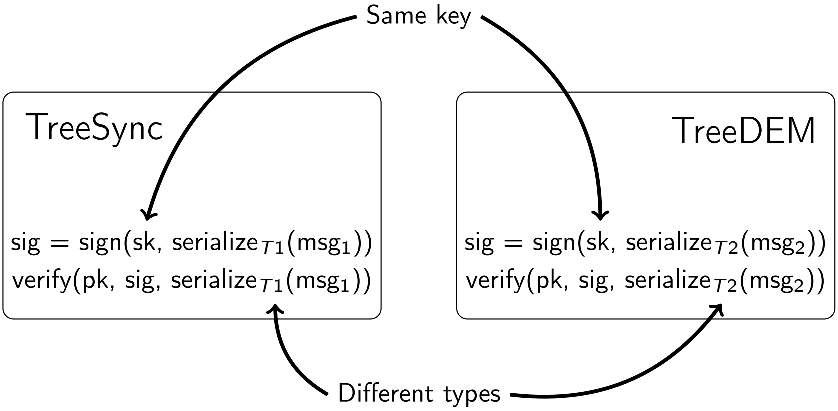
$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T1}(\mathsf{msg}_1))$

TreeDEM

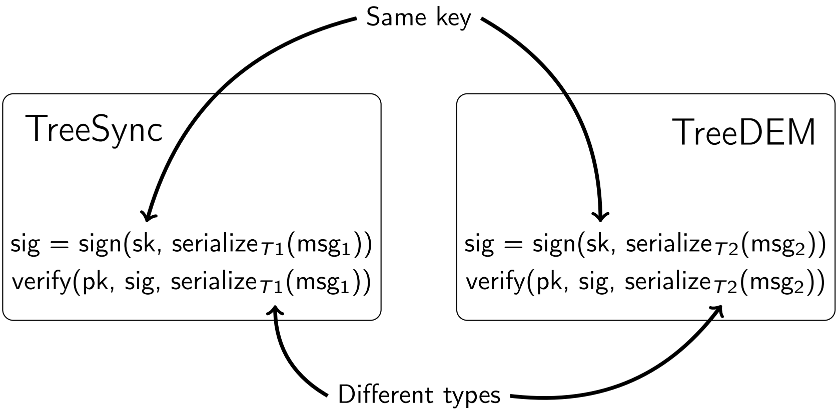
$\mathsf{sig} = \mathsf{sign}(\mathsf{sk}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$

$\mathsf{verify}(\mathsf{pk}, \mathsf{sig}, \mathsf{serialize}_{T2}(\mathsf{msg}_2))$

Contribution: Fixing a signature ambiguity attack

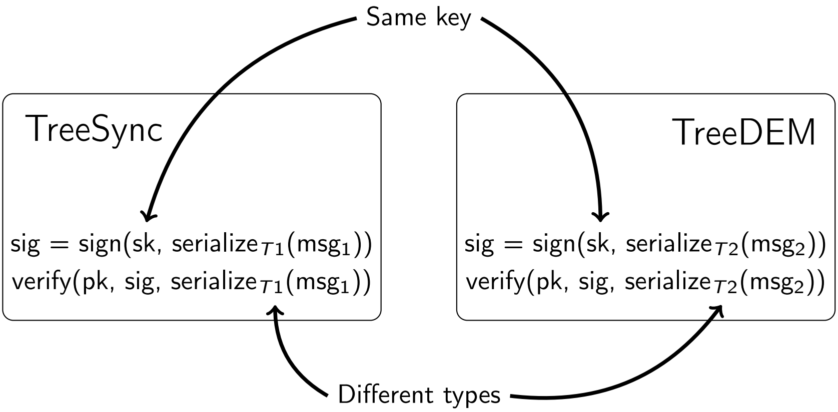


Contribution: Fixing a signature ambiguity attack



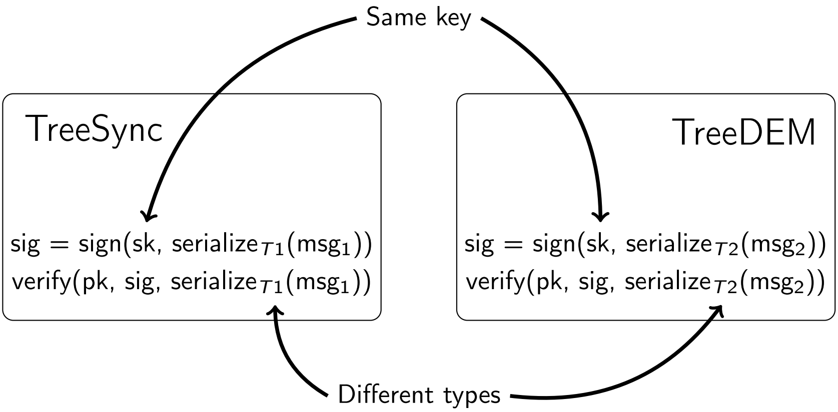
What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?

Contribution: Fixing a signature ambiguity attack



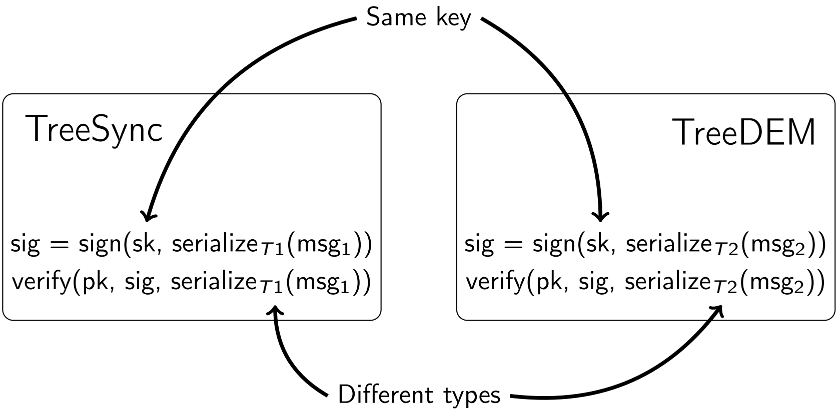
What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?
Bad interaction between TreeSync and TreeDEM!

Contribution: Fixing a signature ambiguity attack



What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?
Bad interaction between TreeSync and TreeDEM!

Contribution: Fixing a signature ambiguity attack



What if $\exists \text{msg}_1 \text{msg}_2, \text{serialize}_{T1}(\text{msg}_1) = \text{serialize}_{T2}(\text{msg}_2)$?
Bad interaction between TreeSync and TreeDEM!

Attack found by doing proofs on a bit-precise specification, thanks to executability and interoperability tests.

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:

$$\begin{array}{ccc} P_1 & & P_2 \\ \uparrow & & \uparrow \\ C_1 & \simeq & C_2 \end{array}$$

Security proof, step 2: the parent-hash guarantee theorem

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

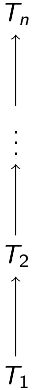
We prove the theorem:

$$\begin{array}{ccc} P_1 & \simeq & P_2 \\ \uparrow & & \uparrow \\ C_1 & \simeq & C_2 \end{array}$$

Security proof, step 3: signature invariant

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

Proof sketch:



Security proof, step 3: signature invariant

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

Proof sketch:

$$
\begin{array}{ccc}
T_n & & T'_n \\
\uparrow & & \uparrow \\
\vdots & & \vdots \\
\uparrow & & \uparrow \\
T_2 & & T'_2 \\
\uparrow & & \uparrow \\
T_1 & \simeq & T'_1
\end{array}
$$

Security proof, step 3: signature invariant

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

Proof sketch:

$$
\begin{array}{ccc}
T_n & & T'_n \\
\uparrow & & \uparrow \\
\vdots & & \vdots \\
\uparrow & & \uparrow \\
T_2 & \simeq & T'_2 \\
\uparrow & & \uparrow \\
T_1 & \simeq & T'_1
\end{array}
$$

Security proof, step 3: signature invariant

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

Proof sketch:

$$
\begin{array}{ccc}
T_n & \simeq & T'_n \\
\uparrow & & \uparrow \\
\vdots & & \vdots \\
\uparrow & & \uparrow \\
T_2 & \simeq & T'_2 \\
\uparrow & & \uparrow \\
T_1 & \simeq & T'_1
\end{array}
$$

Final notes

Proof effort

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

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

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL