Formalizing MLS in F*

(a progress report)

B. Beurdouche, K. Bhargavan, P. Naldurg, T. Wallez

Inria

What have those folks in Inria been doing?

Our Goal: Security proof for a precise *comprehensive* model of MLS

Our Approach:

- Build an executable *interoperable* model of MLS in F*
- Define security goals as typed invariants in F*
- Prove functional correctness and security with byte-level precision
- Refine the executable model into a verified optimized implementation in C

What have those folks in Inria been doing?

Research Reports:

- *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS,* K. Bhargavan, B. Beurdouche, P. Naldurg, INRIA Research Report, 2020 https://hal.inria.fr/hal-02425229
- *Formal Verification for High Assurance Security Software in F*,* B. Beurdouche, PhD Thesis, Inria 2020

What it covers:

- Formal models in F* of ART, mKEM, TreeKEM (+ Blanking + Tree Signatures)
- Malicious insiders, Double Join attacks (on Add, Remove, and Joiner)
- Inductive proofs of symbolic security properties for any group size
- Requires a new symbolic verification framework for F* to handle FS, PCS

DY*: Symbolic Proofs for Crypto Protocol Code in F*

Research Report:

- *DY* : A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code,* K. Bhargavan, A. Bichhawat, Q. Do, P. Hosseyni, R. Küsters, G. Schmitz, T. Würtele, Euro S&P 2021, https://hal.inria.fr/hal-03178425/

What it is:

- A new framework for symbolically verifying cryptographic protocols in F*
- Design influenced by MLS, but applicable more generally
- A novel trace-based semantics allows clean formulation of FS and PCS
- Modular security proofs for recursive data structures, composite protocols
- Soundness of symbolic verification proved within F*
- Applied to Signal, Noise, ACME, … and now to MLS

MLS*: A Formal Model of MLS draft-11 in F*

Ongoing Work (Théophile Wallez, Benjamin Beurdouche, Karthik Bhargavan):

- An interoperable model of draft-11 in F*
- A modular specification decomposed into several sub-protocols
- Shares and passes test vectors from OpenMLS
- Uses HACL* for underlying cryptography
- ~ 2K lines of F*

Next Steps:

- Encode security goals using DY*
- Prove modular symbolic security for the full MLS spec
- Expect public release in August 2021

Some feedback and questions from the formalization

1. Decomposing MLS into sub-protocols for modular proofs
2. Optimizing unmerged leaves to lower tree size to O(n)
3. Simplifying KeyPackage and its extensions
4. Understanding the need for Tree Math

Decomposing MLS

MLS is getting pretty large

A monolithic protocol that has evolved with many goals:

- Managing dynamic group membership
- Distributing group keys with FS and PCS (Ratchet Trees)
- Encrypting messages with FS (Message Framing)
- Preventing insider attacks (Blanking, Parent hash)
- Optimizations (Unmerged leaves)

Hard to understand or reason about full protocol in one-shot:

- Would be nice to decompose the protocol to enable modular proofs

TreeSync, TreeKEM, and TreeDEM

TreeSync

- A tree-based dynamic group management and synchronization protocol
- Treats key packages (node/leaf content) as opaque bytestrings
- Ensures tree agreement and authenticity, using signatures and hashing

TreeKEM

- A tree-based group key distribution protocol
- Assumes authenticated tree (from TreeSync)
- Ensures FS and PCS for node/epoch/init secrets, using HPKE and KDF

TreeDEM

- A tree-based application message encryption protocol
- Assumes authenticated tree (from TreeSync) and epoch secrets (from TreeKEM)
- Ensures authenticity and FS for messages, using AEAD, KDF, and signatures

Separating TreeSync from TreeKEM

TreeSync

- Does not care about encryption/secrecy/key derivation
- Handles create, add, remove, + blanking, unmerged leaves, parent hash
- Focuses on data structure integrity against outsiders and insiders
- All double join attacks can be demonstrated directly on TreeSync
- Enforces a “write” access control policy on the ratchet tree

Tree Authentication Invariant

- The content at a non-blank node “n” must have been written by one of the members at some leaf “l” under node “n” (at some prior epoch “i”)
  - Relies on tree signing to guarantee sub-tree integrity
  - If the signing key at leaf “l” was uncompromised at epoch “i”,
    then the subtree is authentic (i.e. it is the same as the subtree at member “l”)

Separating TreeKEM from TreeSync

TreeKEM

- Does not care about signatures/authentication/parent hash
- Focuses on group key derivation
- Enforces a “read” access control policy on the ratchet tree secrets

Tree Secrecy Invariant

- The node secret at a non-blank node “n” can only be read by one of the members at some leaf “l” under node “n” (at current epoch “i”)

```
The private key for a node in the tree is known to a member of
the group only if that member's leaf is a descendant of the node.
```

Benefits of Decomposition

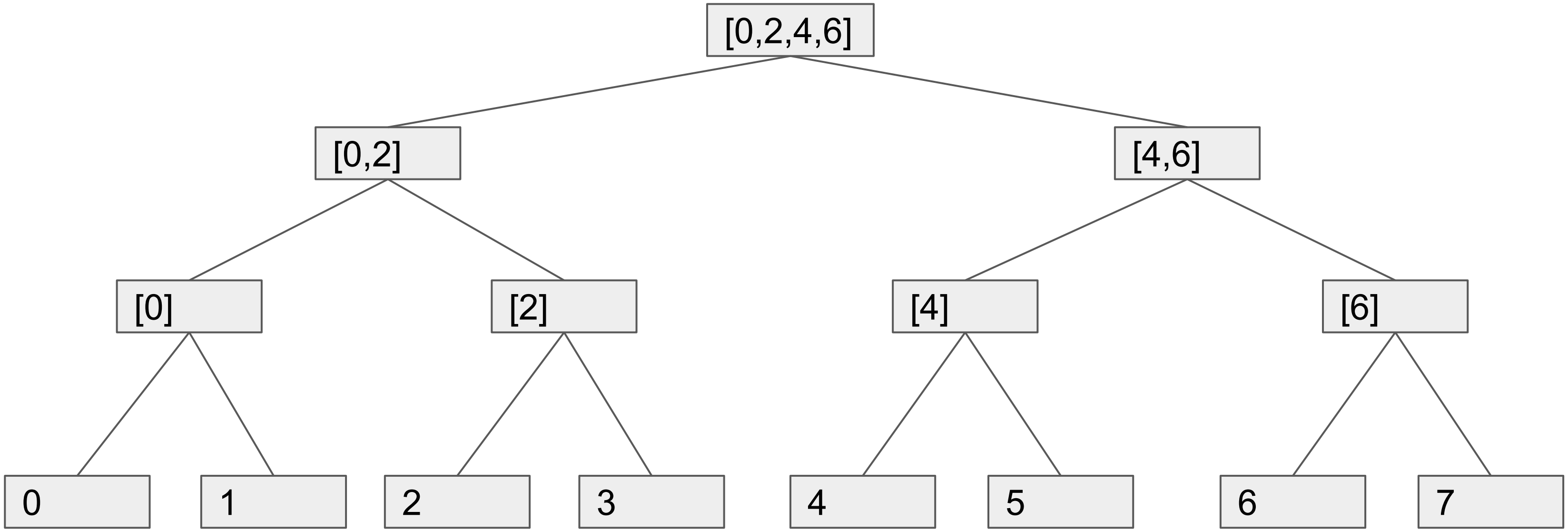
- Dividing up the protocol makes modular specification easier
- In our spec: TreeSync - 400 lines, TreeKEM - 350 lines, TreeDEM - 200 lines
- We expect that our security proofs will also be more modular (TBD)
- Perhaps this decomposition also helps understanding and implementation?
- Shall we make it explicit in the RFC?

Improving Modularity

- Some features break modularity and uglify our spec
- Parent Hash (TreeSync) explicitly relies on sibling Tree Resolution (TreeKEM)
- We can remove this dependency by using sibling Tree Hash instead.

Optimizing the unmerged leaves design

The current unmerged leaves design: O(n log n) tree size

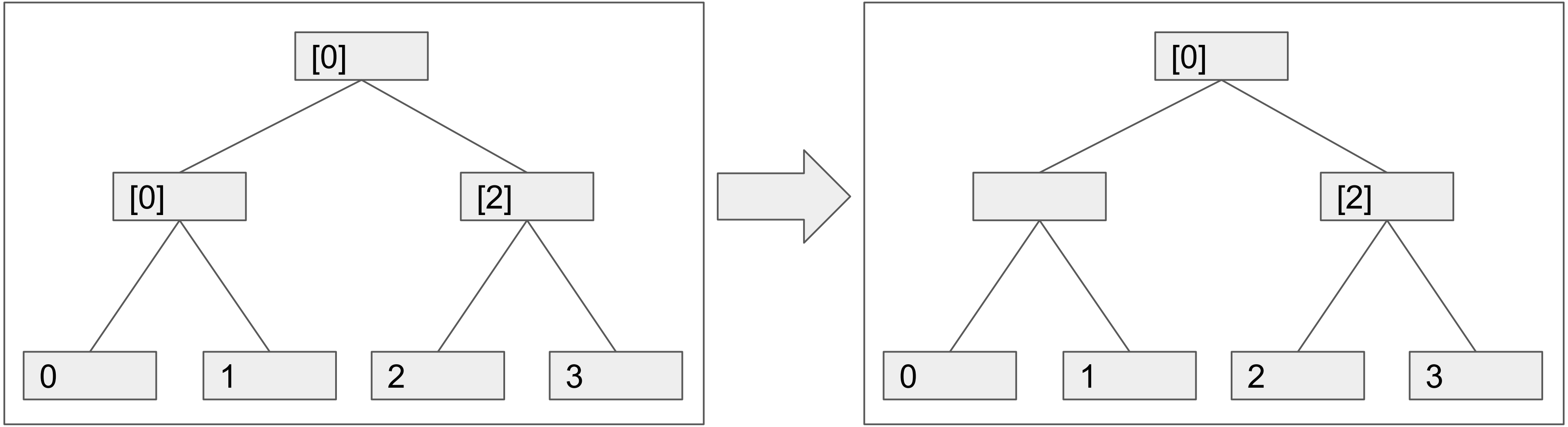


Solution 1: a patch on the current design

Invariant:

`leaf` is a descendent of `node`
and `leaf` in `parent(node).unmerged_leaves`
⇒ `leaf` in `node.unmerged_leaves`

Solution: only store `leaf` in the highest node possible



Solution 1: a patch on the current design

Pros:

- Straightforward fix
- Bring back the size of the tree to O(n)

Cons:

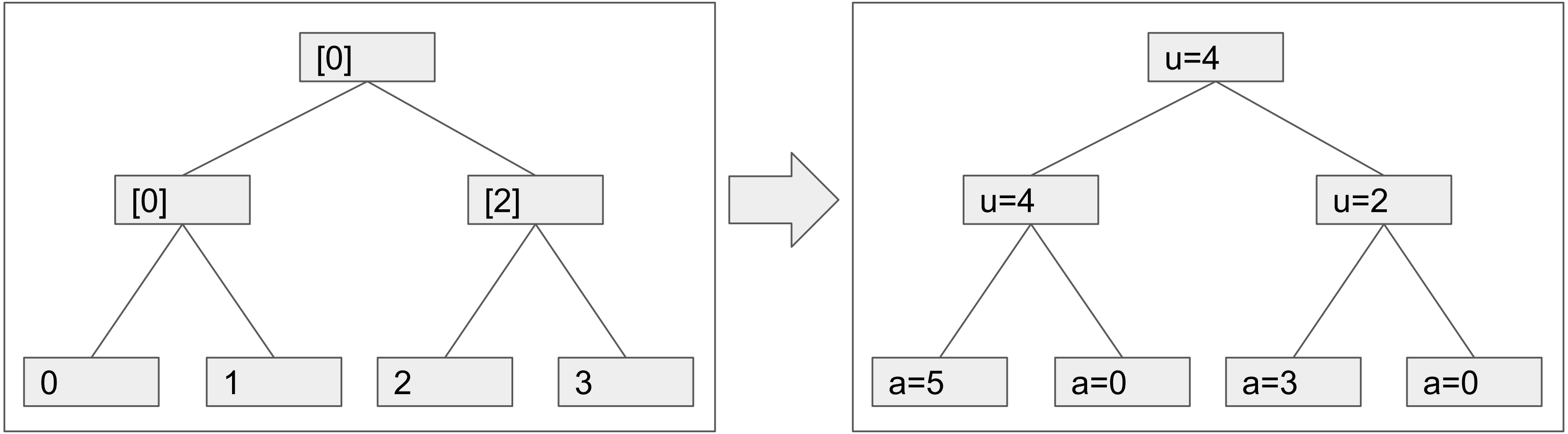
- Makes the RFC harder to understand, the implementation more bug-prone
- More nodes are changed when processing an UpdatePath (nodes on path + copath) [Raphael Robert]

Solution 2: a new design using version numbers

Store epoch numbers in parent nodes and leaves:

- In leaves: store the epoch when the leaf was added
- In nodes: store the epoch of last UpdatePath going through this node

Property: `leaf` in `node.unmerged_leaves` ⇔ `node.last_update_epoch < leaf.add_epoch`



Solution 2.1: a new design using version numbers

Observation 1:

```
node.last_update_epoch < leaf.add_epoch
⇔
node.last_update_epoch < leaf.last_update_epoch
```

Consequence 1: we can store last update epoch everywhere.

Observation 2:

```
node.last_update_epoch = left(node).last_update_epoch
```

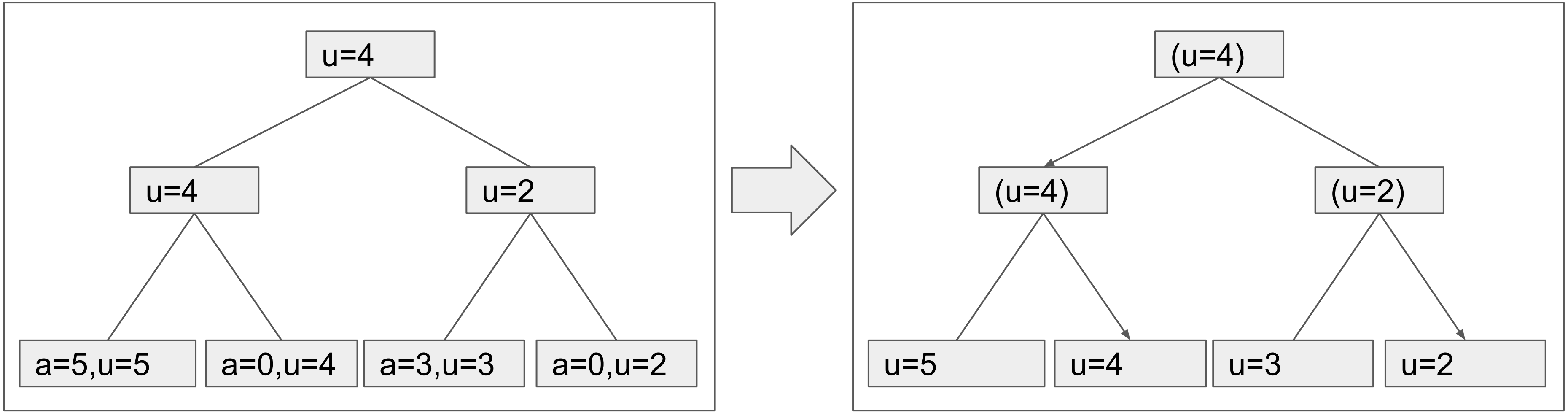
or

```
node.last_update_epoch = right(node).last_update_epoch
```

(it depends on the direction of the last UpdatePath going through that node)

Consequence 2: we can only store this direction in parent nodes.

Solution 2.1: a new design using version numbers



Solution 2.1: a new design using version numbers

Pros:

- Bring back the size of the tree to O(n)
- Easy to understand
- Add useful information to the tree

Cons [Raphael Robert]:

- The last update epoch will be public, this could be useful for an attacker?
- On small trees / trees with few unmerged leaves, this actually increases the size

Simplifying KeyPackage

Looking inside KeyPackage

```
struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey hpke_init_key;
    Credential credential;
    Extension extensions<8..2^32-1>;
    opaque signature<0..2^16-1>;
} KeyPackage;
```

Should be parameters of the group? (version, cipher_suite)

Not an init key (just a key) (hpke_init_key)

Contain important data!
(e.g. parent hash) (extensions)

The history of KeyPackage

KeyPackage is used for two things:

- Init key: key that is published beforehand to add new members
- Leaf content: all the public information needed for a leaf in the tree

KeyPackage is constructed as an InitKey: what is useful for leaf content is put in extensions.

Solution: make two separate structures for init keys and leaf contents?

Understanding the need for Tree Math

(and the concept of “node index”)

Where is Tree Math needed?

- To describe operations on the ratchet tree
- Tree hash
- Secret tree
- Ratchet tree extension

Tree Math is language agnostic, but it bakes in an implementation strategy.

Is it really needed? Could we do things more abstractly?

Removing Tree Math to describe operations on ratchet tree

Give an API that allow to work on left-balanced binary trees:

- `root(tree)`
- `left(node)`
- `right(node)`
- `parent(node)`
- `is_leaf(node)`

Removing Tree Math in Tree Hash

Current:

```
struct {
    uint32 node_index;
    optional<KeyPackage> key_package;
} LeafNodeHashInput;
struct {
    uint32 node_index;
    optional<ParentNode> parent_node;
    opaque left_hash<0..255>;
    opaque right_hash<0..255>;
} ParentNodeTreeHashInput;
```

Suggestion:

```
struct {
    uint32 leaf_index; //or remove it?
    optional<KeyPackage> key_package;
} LeafNodeHashInput;
struct {
    //node_index removed
    optional<ParentNode> parent_node;
    opaque left_hash<0..255>;
    opaque right_hash<0..255>;
} ParentNodeTreeHashInput;
```

Removing Tree Math in Secret Tree

Current

```
tree_node_[N]_secret
|
|
+--> DeriveTreeSecret(., "tree", left(N), 0, KDF.Nh)
|    = tree_node_[left(N)]_secret
|
+--> DeriveTreeSecret(., "tree", right(N), 0, KDF.Nh)
     = tree_node_[right(N)]_secret
```

Node index (uint32)

Suggestion:

```
tree_node_[N]_secret
|
|
+--> DeriveTreeSecret(., "tree", 0, 0, KDF.Nh)
|    = tree_node_[left(N)]_secret
|
+--> DeriveTreeSecret(., "tree", 1, 0, KDF.Nh)
     = tree_node_[right(N)]_secret
```

“Abstract” node

Removing Tree Math in the ratchet tree extension

Simply say that it is serialized in infix order!

It is possible to reconstruct the tree only with `log2` and `pow2`.