A symbolic analysis of MLS in F*

Karthikeyan Bhargavan
Benjamin Beurdouche
Prasad Naldurg
MLS Interim, Jan 2020

What we did

- Formal requirements for MLS as an F* interface
  - Functional and security goals written *independent* from protocol mechanisms
- Formal specification for MLS draft 7 as an F* module
  - **Covers:** key establishment, long-term signature keys, message protection
  - **Does not cover:** application tree key schedule, proposals/commits, ...
- Proof that the specification meets the requirements
  - Assuming a symbolic model of cryptographic primitives
  - Machine-checked proof by type-checking in F*

Specifying MLS draft 7 in F*

A succinct, modular, executable, formal specification

- **Succinct**: A spec of MLS in ~200 lines of F*
- **Modular**: Separation between key establishment / message protection.
- **Executable:** The spec can be executed to obtain concrete traces, and can be used as an interop target
- **Easy to modify:** By changing 2-3 functions, we obtain specs for mKEM, ART, TreeKEM (without blanking), TreeKEM (with signatures) etc.

Modeling Compromise

- Attacker can compromise any member’s credential
- Attacker can compromise the current decryption key of a member
- Attacker can compromise a previous decryption key of a member
- Using any of these compromises, attacker can try to actively attack the protocol

- Modeling this level of **fine-grained, dynamic, active compromise** in F* required a new model of global state and corruption events

Verified Security Goals

- The **membership** of a group state g is the *current versions* of the *current members* of g
- **Message Confidentiality:** a message sent in group state g is confidential as long as the decryption keys of all members of g remain uncompromised
- **Message and Sender Authenticity:** a message received from some sender in group state g is authentic if the current credential of the sender is uncompromised
- **Group Agreement:** after receiving a group operation, the group state at the receiver and sender is the same, if the sender's credential is uncompromised
- Applying Update, Add, Remove operations preserves these properties

What about FS, PCS?

- FS and PCS are imprecise terms when used with groups, needs context
- In our model, by looking at the membership of a group state, one can *read* off various security guarantees that can be interpreted as specific forms of FS and PCS

- **Update FS:** If a member updates its key and the new key is then compromised, the previous group secret remains confidential (rTreeKEM provides a strictly stronger guarantee)
- **Add FS:** If a member is added and is then compromised, the previous group secret remains confidential
- **Update PCS:** If a member updates its key and the previous key is then compromised, the next group secret remains confidential

A Note on Remove PCS

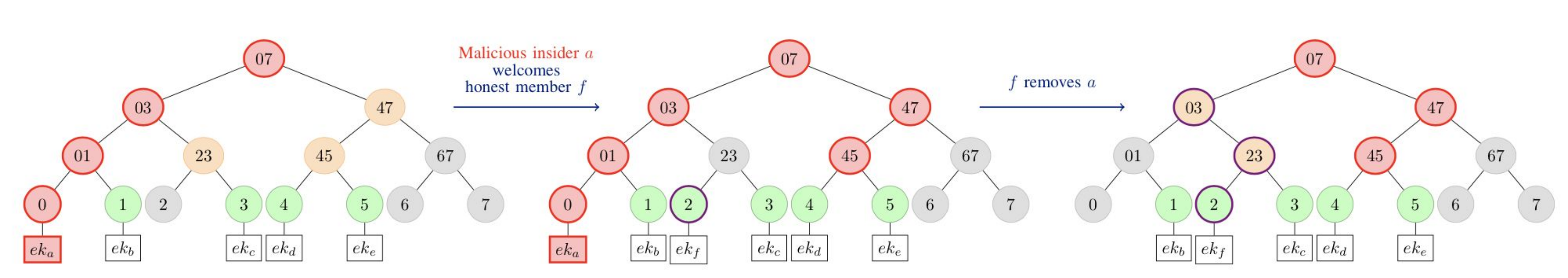
- In two party protocols, we usually only consider recovery against *passive* compromise, so PCS is really seen as post *passive* compromise security
- In group messaging, we have another choice: we could *remove* the misbehaving member, to obtain post *active* compromise security

- **Remove Security:** If a member was (actively) compromised and is then removed from a group, the new group secret remains confidential

  (We believe this should be an explicit goal of MLS protocols.)

Attacks that appeared in our model

- **Double Join:** our model of TreeKEM (without blanking) has a double-join attack; TreeKEM (with blanking) has a double-join attack at Welcome



- **Cross-Group Forwarding Attack:** Draft 7 does not include transcript in message signatures, enabling this attack
- **Stream Truncation between epochs:** no previous-message counter

Ongoing and Future Work

- Release models along with F*-based analysis framework
- Interop test our specification with other implementations

- Incorporate proposals+commits, application key schedule, etc.
- Experiment with plugging in UPKE to obtain a model of rTreeKEM
- Experiment with rotating signature keys, tree signatures, …

- Link our symbolic proofs to a computationally sound crypto model

Formalizing Requirements: Members

```
(* Public Information about a Group Member *)
type member_info = {
  cred: credential;
  version: nat;
  current_enc_key: enc_key}
```

- Members have *versioned* encryption keys (tracked within the protocol)
- Members have credentials that can be validated by any other member (Credentials may also change over time)
- An attacker can compromise a specific *version* of a specific *member*

An API for Group Management

```
(* Group State Data Structure *)
val group_state: datatype
val group_id: group_state → nat
val max_size: group_state → nat
val epoch: group_state → nat
type index (g:group_state) = i:nat{i < max_size g}
type member_array (sz:nat) =
    a:array (option member_info){length a = sz}
val membership: g:group_state → member_array (max_size g)
```

```
(* Create a new Group State *)
val create: gid:nat → sz:pos → init:member_array sz
        → entropy:bytes → option group_state
(* Group Operation Data Structure *)
val operation: datatype
(* Apply an Operation to a Group *)
val apply: group_state → operation → option group_state
(* Create an Operation *)
val modify: g:group_state → actor:index g
        → i:index g → mi':option member_info
        → entropy:bytes → option operation
(* Group Secret shared by all Members *)
val group_secret: datatype
(* Calculate Group Secret *)
val calculate_group_secret: g:group_state → i:index g
        → ms:member_secrets → option group_secret
        → option group_secret
```

- Abstract types for groups, operations
- Each protocol instantiates it with its own data structures
- Each protocol then implements: create, apply, modify, calculate_secret

An API for Message Protection

```
(* Protocol Messages *)
type msg =
    | AppMsg: ctr:nat → m:bytes → msg
    | Create: g:group_state → msg
    | Modify: operation → msg
    | Welcome: g:group_state → i:index g
            → secrets:bytes → msg
    | Goodbye: msg
(* Encrypt Protocol Message *)
val encrypt_msg: g:group_state → gs:group_secret
      → sender:index g → ms:member_secrets → m:msg
      → entropy:bytes → (bytes * group_secret)
(* Decrypt Initial Group State *)
val decrypt_initial: ms:member_secrets
      → c:bytes → option msg
(* Decrypt Protocol Message *)
val decrypt_msg: g:group_state → gs:group_secret
      → receiver:index g → c:bytes
      → option (msg * sender:index g * group_secret)
```

- Multiple kinds of message
- Message protection is independent of group key establishment, except that it relies on the group secret

```
(* Definition of the main data structures in TreeKEM_B *)
type member_secrets = {
      identity_sig_key: sign_key;
      leaf_secret:secret;
      current_dec_key: dec_key}
type direction = | Left | Right
type key_package = {
      from : direction;
      node_enc_key: enc_key;
      node_ciphertext: bytes}
type group_state = {
      group_id: nat;
      levels: nat;
      tree: tree levels;
      epoch: nat;
      transcript_hash: bytes}
let index_l (l:nat) = x:nat{x < pow2 l}
type operation = {
      lev: nat;
      index: index_l lev;
      actor: credential;
      path: path lev & path lev}
type group_secret :eqtype = {
      init_secret: secret; hs_secret: secret; sd_secret: secret;
      app_secret: secret; app_generation: nat}
```

```
(* Auxiliary tree λctions *)
let child_index (l:pos) (i:index_l l) : index_l (l−1) & direction =
  if i < pow2 (l − 1) then (i,Left) else (i−pow2 (l−1),Right)
let key_index (l:nat) (i:index_l l) (sib:tree l) dir : index_l (l+1) =
  if dir = Left then i else i + length (pub_keys l sib)
let order_subtrees dir (l,r) = if dir = Left then (l,r) else (r,l)
```

```
(* Create a new tree from a member array *)
let rec create_tree (l:nat) (c:credential)
                    (init:member_array (pow2 l)) =
  if l = 0 then Leaf c init.[0]
  else let init_l,init_r = split init (pow2 (l−1)) in
       let left = create_tree (l−1) c init_l in
       let right = create_tree (l−1) c init_r in
       Node c None left right
(* Apply a path to a tree *)
let rec apply_path (l:nat) (i:nat{i<pow2 l}) (a:credential)
                   (t:tree l) (p:path l) : tree l =
  match t,p with
  | Leaf _ m, PLeaf m' → Leaf a m'
  | Node _ _ left right, PNode nk next →
      let (j,dir) = child_index l i in
      if dir = Left
      then Node a nk (apply_path (l−1) j a left next) right
      else Node a nk left (apply_path (l−1) j a right next)
(* Create a blank path after modifying a leaf *)
let rec blank_path (l:nat) (i:index_l l)
                   (mi:option member_info) : path l =
  if l = 0 then PLeaf mi
  else let (j,dir) = child_index l i in
       PNode None (blank_path (l−1) j mi)
(* Create an update path from a leaf to the root *)
let rec update_path (l:nat) (t:tree l) (i:nat{i<pow2 l})
                    (mi_i:member_info) (s_i:secret)
                    : option (path l & s_root:secret) =
  match t with
  | Leaf _ None → None
  | Leaf _ (Some mi) → if name(mi.cred) = name(mi_i.cred)
                       then Some (PLeaf (Some mi_i),s_i)
                       else None
  | Node _ _ left right →
      let (j,dir) = child_index l i in
      let (child,sibling) = order_subtrees dir (left,right) in
      match update_path (l−1) child j mi_i s_i with
      | None → None
      | Some (next, cs) →
        let ek_sibling = pub_keys (l−1) sibling in
        let kp,ns = node_encap cs dir ek_sibling in
        Some (PNode (Some kp) next, ns)
```

```
(* Create a new group state *)
let create gid sz init leaf_secret =
  match init.[0], log2 sz with
  | None,_ → None
  | _,None → None
  | Some c,Some l →
    let t = create_tree l c.cred init in
    let ek = pk leaf_secret in
    let mi' = {cred=c.cred; version=1; current_enc_key=ek} in
    (match update_path l t 0 mi' leaf_secret with
    | None → None
    | Some (p,_) → let t' = apply_path l 0 c.cred t p in
                   let g0 = {group_id=gid; levels=l;
                             tree=t'; epoch=0;
                             transcript_hash = empty} in
                   let h0 = hash_state g0 in
                   Some ({g0 with transcript_hash = h0}))
(* Apply an operation to a group state *)
let apply g o =
  if o.lev ≠g.levels then None
  else let p1,p2 = o.path in
       let t' = apply_path o.lev o.index o.actor g.tree p1 in
       let t' = apply_path o.lev o.index o.actor g.tree p2 in
       Some ({g with epoch = g.epoch + 1; tree = t';
              transcript_hash = hash_op g.transcript_hash o})
(* Create an operation that modifies the group state *)
let modify g actor i mi_i leaf_secret =
    match (membership g).[actor] with
    | None → None
    | Some mi_a_old →
      let mi_a = update_member_info mi_a_old leaf_secret in
      let bp = blank_path g.levels i mi_i in
      let nt = apply_path g.levels i mi_a.cred g.tree bp in
      match update_path g.levels nt actor mi_a leaf_secret with
      | None → None
      | Some (up,_) → Some ({lev = g.levels; actor= mi_a.cred;
                             index = i; path = (bp,up)})
```

```
(* Calculate the subgroup secret for the root of a tree *)
let rec root_secret (l:nat) (t:tree l) (i:index_l l) (leaf_secret:bytes)
      : option (secret & j:nat{j < length (pub_keys l t)}) =
  match t with
  | Leaf _ None → None
  | Leaf _ (Some mi) → Some (leaf_secret, 0)
  | Node _ (Some kp) left right →
    let (j,dir) = child_index l i in
    let (child,_) = order_subtrees dir (left,right) in
    (match root_secret (l−1) child j leaf_secret with
    | None → None
    | Some (cs,i_cs) →
      let (_,recipients) = order_subtrees kp.from (left,right) in
      let ek_r = pub_keys (l−1) recipients in
      (match node_decap cs i_cs dir kp ek_r with
       | Some k → Some (k,0)
       | None → None))
  | Node _ None left right →
    let (j,dir) = child_index l i in
    let (child,sib) = order_subtrees dir (left,right) in
    match root_secret (l−1) child j leaf_secret with
    | None → None
    | Some (cs,i_cs) → Some (cs,key_index (l−1) i_cs sib dir)
(* Calculate the current group secret *)
let calculate_group_secret g i ms gs =
  match root_secret g.levels g.tree i ms.leaf_secret with
  | None → None
  | Some (rs,_) →
    let prev_is = if gs = None then empty_bytes
                  else (Some?.v gs).init_secret in
    let (apps,hs,sds,is') =
        derive_epoch_secrets prev_is rs g.transcript_hash in
    Some ({init_secret = is';hs_secret = hs; sd_secret = sds;
           app_secret = apps; app_generation = 0})
```

Subgroup Secrecy Invariant

- Every occupied leaf in the tree with member info mi contains a valid credential with a verification key labeled with the auth session of mi. Further, the current encryption key at the leaf is labeled with the dec session of mi.

- Every non-blank node in the tree contains a key package with a public encryption key and a ciphertext. If none of the members of the sub-tree are auth compromised, then the label of the encryption key matches the tree label of the subtree, and the ciphertext contains an encrypted secret that is also labeled with the tree label of the subtree