Malicious Hubs

draft-burger-mimi-audit-layer-00

Eric Burger
eburger@standardstrack.com

Motiviation

- MIMI relies on a trusted hub for routing and timestamping messages
- MLS provides End-to-End Encryption (E2EE), but hub is critical attack surface
- Even with E2EE, compromised hub can:
  - Reorder messages
  - Drop Messages

Example: Message Dropping

As seen by Charlie

- 08:00 Alice: *Should we buy MumbleCo?*
- 8:02 Bob: *Buy MumbleCo – it feels good.*
- 08:03 Charlie: *Do not buy MumbleCo, the research says they will go bankrupt!*
- 08:08 Alice: *We bought MumbleCo and it went out of business!*
- 08:09 Charlie: *I told you not to buy MumbleCo, but you went with Bob's suggestion on his feelings???*

As seen by Alice

- 08:00 Alice: *Should we buy MumbleCo?*
- 08:02 Bob: *Buy MumbleCo – it feels good.*
- 08:08 Alice: *We bought MumbleCo and it went out of business!*
- 08:09 Charlie: *I told you not to buy MumbleCo, but you went with Bob's suggestion on his feelings???*

- **Impact:** Misunderstanding and undermined chat integrity.

Threat Model

- Nation state actor or insider threat
- Can compromise hub
- Hub can do traffic analysis to detect ‘checking’
- Can coordinate compromised client and hub to defeat E2EE
  - Hub *can* determine which messages could have an impact
  - [hat tip to Rohan]

Threats Addressed

1. Timestamp Authority:
   Ability to backdate or falsify timestamps.
2. Re-ordering via Delay:
   Selectively delaying messages to cause arbitrary permutations.
3. Isolated Views:
   Clients rely solely on the hub, lacking cross-verification means.

Client-Driven Audit Layer

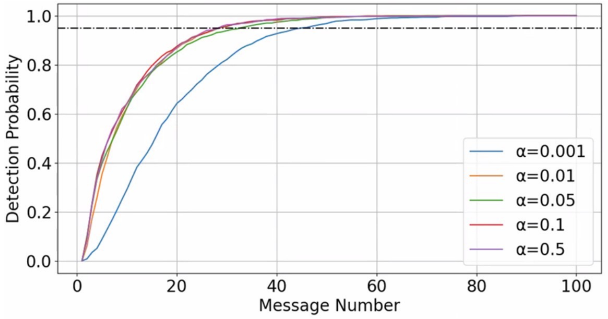
- Use Merkle-tree-based proofs to detect malicious hub behavior
- Key Features:
  - Client-driven: No modification to the hub, no new trusted servers required.
  - Minimal communication overhead.
  - No change to MIMI or MLS
- High-Level Flow:
  - Clients maintain ordered list of messages and timestamps
  - Randomly generate and broadcast Merkle proofs (currently encrypted within regular messages)
  - Receiving clients verify proofs against local message lists
  - Mismatch triggers an alert broadcast

Client Verification & Detection

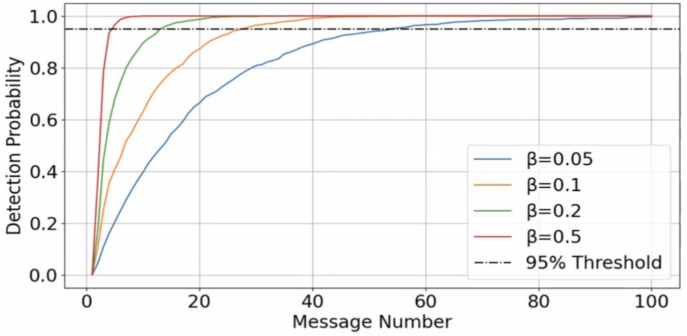
- Verification Process
  - Client computes its own Merkle root (R_hat) from messages s to e.
- Detection
  - If R_hat ≠ R || local t_max ≠ received t_max, a discrepancy is found.
- Alert
  - Broadcast a message to all clients, indicating a malicious hub
  - Looks to hub like any normal application message

Rate-Limiting Proof Messages

- Balance detection with overhead
- Proof-request probability per message: $P_{proof} = 1 - (1 - \alpha)^n$
  - n: number of clients
  - α: individual client's probability of requesting a proof
- Detection probability for T messages: $P^T_{detect} = 1 - [1 - \beta\,(1 - (1 - \alpha)^n)]^T$
  - β: probability of a malicious hub attacking each message
- And doing traffic obfuscation to defeat traffic analysis



(β = 0.10, n = 100)



(α = 0.05, n = 100)

α ≈ 0.05 yields 95% detection probability within T = 40 messages when group size n = 100 and hub-attack probability β = 0.10.

Adjust for larger groups or different risk tolerances

Comparison of Franking, AppAck, and Audit Layer

	Franking	AppAck	Audit Layer
Purpose	Ensures sender cannot deny sending message to a particular group	Confirms message range has been received	Detects malicious hub actions such as message reordering, dropping, or inconsistent application state
Mechanism	Hub adds accepted_timestamp & sender URI to message & signs; clients notice if timestamp looks wonky	Client submits MLS Proposal with message range seen; at MLS Commit time (epoch) all clients will know if state is wonky	Clients create Merkle proofs of timestamped message streams, probabilistically share them, and verify against peers' views
Comparison	Might catch selective dropping and reordering	Relies on luck the client doing the Proposal has seen an anomaly, the client doing the Commit sees the right proposal, and the client doing the Commit is honest	**At application layer** Feature: if your application does not care about risk, do not need to do it Bug: requires enough clients to implement it to catch bad hub

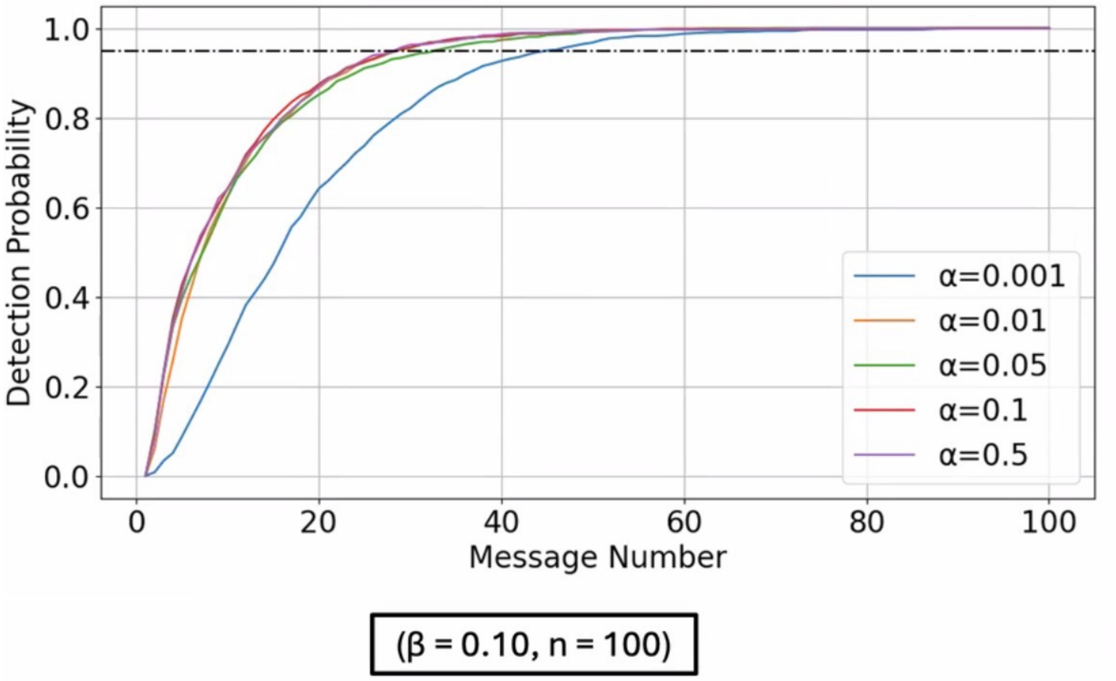
Open Issues

- Compromised Client
  - A compromised client can generate fake proofs, leading to false proof mismatch broadcasts.
  - Several mitigation strategies to examine.
- Examine integrating with AppAck
  - Like the property of using MLS state
  - Need to work out consensus mechanism
    - vs. whomever does the Commit first

Discussion

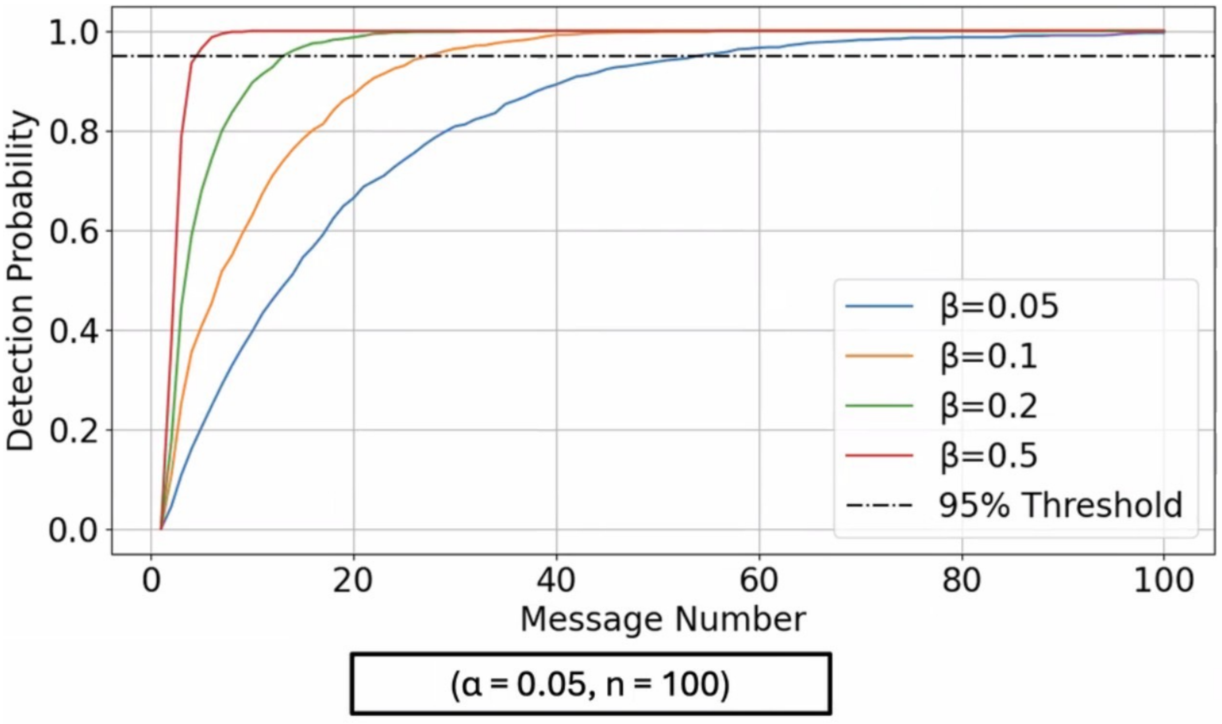
Backup Slides

Impact of Client Sampling Rate α



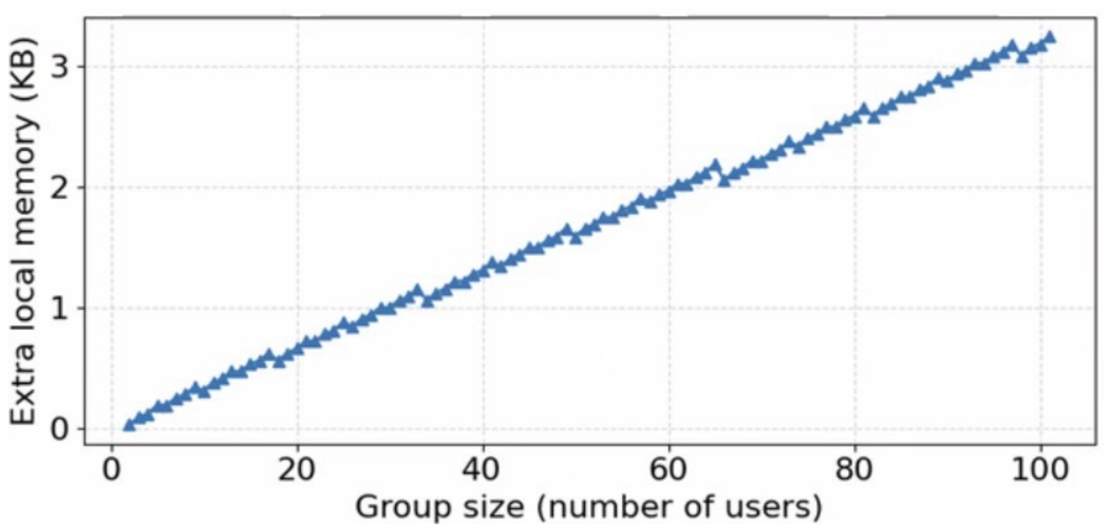
- Performed evaluation using 1000 trial Monte-Carlo simulations with varying group sizes till 100 nodes.
- **Low a (0.001): H**igh detection latency (> 60 messages for 95% detection).
- **Moderate/High a (0.01 - 0.05): S**ignificant improvement in detection speed (optimal point).
- **High a (>0.05):** Diminishing returns.

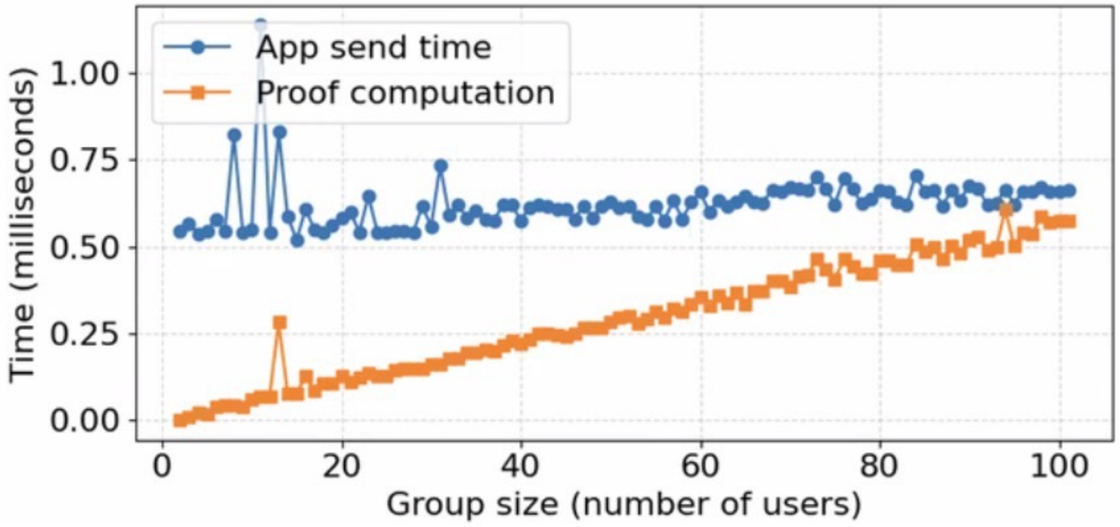
Impact of Client Sampling Rate β



- **High β:** Faster detection, as attacks occur more frequently.
- **Low β:** Longer to detect due to infrequent attacks.

Computation Overhead





- **Memory overhead** increases linearly with group size and is ~3kB for 100 users.
- **Latency overhead** for proof computation increases linearly with group size and is ~0.6ms

Merkle Tree Generation and Broadcast

- **Client State:** Each client maintains a list of messages $M = [(d_i, t_i)]$ sorted by hub-assigned timestamps.
  - d_i: Message plaintext
  - t_i: Hub-assigned timestamp
- **Leaf Node Generation:** $l_i = H(d_i \| t_i)$
- **Tree Construction:**
  - Batch of messages (from index s to e).
  - Parent node: $n_j = H(l_k \| l_{k+1})$.
  - Recursively combine until a single Merkle Root R is generated.
  - Handle odd number of leaves by duplicating the last node.
- **Merkle Proof Tuple:** (R, t_{max}, s, e)

Merkle Proof Generation and Broadcast Algorithm

Algorithm 1 ComputeAndBroadcastMerkleProof(L_m, s, e)

Require: List of sorted messages $L_m = M[1, \ldots, m]$, indices $1 \leq s \leq e \leq m$
Ensure: Proof tuple $(R, t_{\max}, s, e)$

```
procedure ComputeMerkleProof(M, s, e)
    leaves ← []
    t_max ← 0
    for i ← s to e do
        (d_i, t_i) ← M[i]
        ℓ_i ← H(d_i || t_i)
        leaves.append(ℓ_i)
        t_max ← max(t_max, t_i)
    end for
    R ← BuildMerkleTreeRoot(leaves)
    return (R, t_max, s, e)
end procedure
procedure BroadcastProof(α)
    if Bernoulli(α) then
        Broadcast(R, t_max, s, e)
    end if
end procedure
```

Merkle Proof Verification and Mismatch Detection

Algorithm 2 Proof Verification and Broadcast

procedure OnProof($R, t_{\max}, s, e$)
    compute leaves $\ell_i = H(d_i \| t_i)$ for $i \in [s, e]$
    $\hat{R} \leftarrow \text{ComputeRootFromLocalMessages}(s, e)$
    if $\hat{R} \neq R$ **or** $t_{\max} \neq \max_{i \in [s,e]} t_i$ **then**
        RaiseAlert(ProofMismatch)
        Broadcast(ProofMismatch)
    end if
end procedure