Malicious Hubs

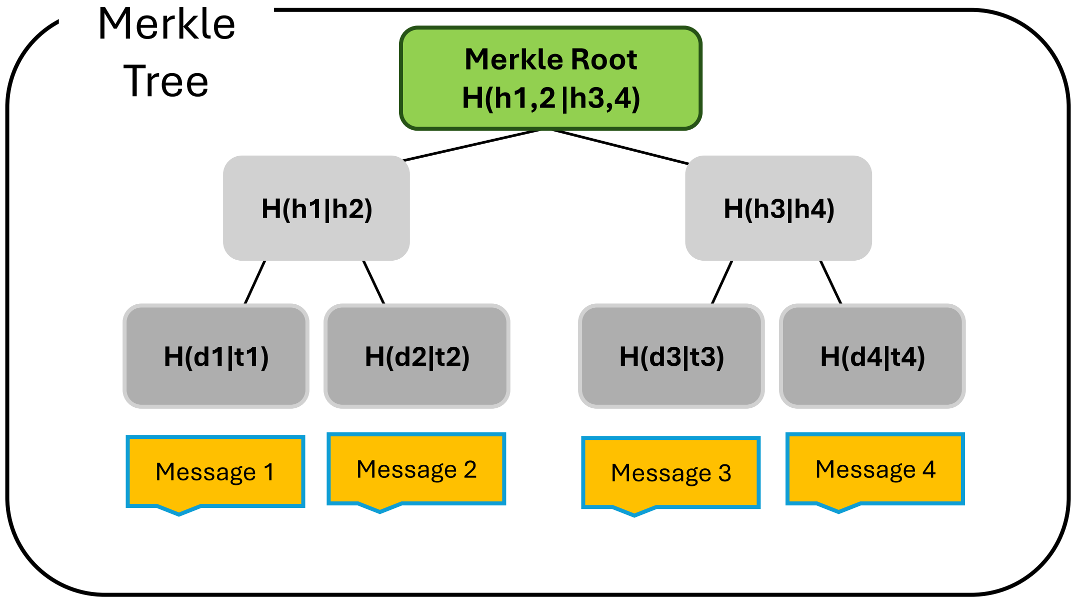
draft-burger-mimi-audit-layer-00

Srinidhi Amuri
srinidhiamuri@vt.edu

Client-Driven Audit Layer

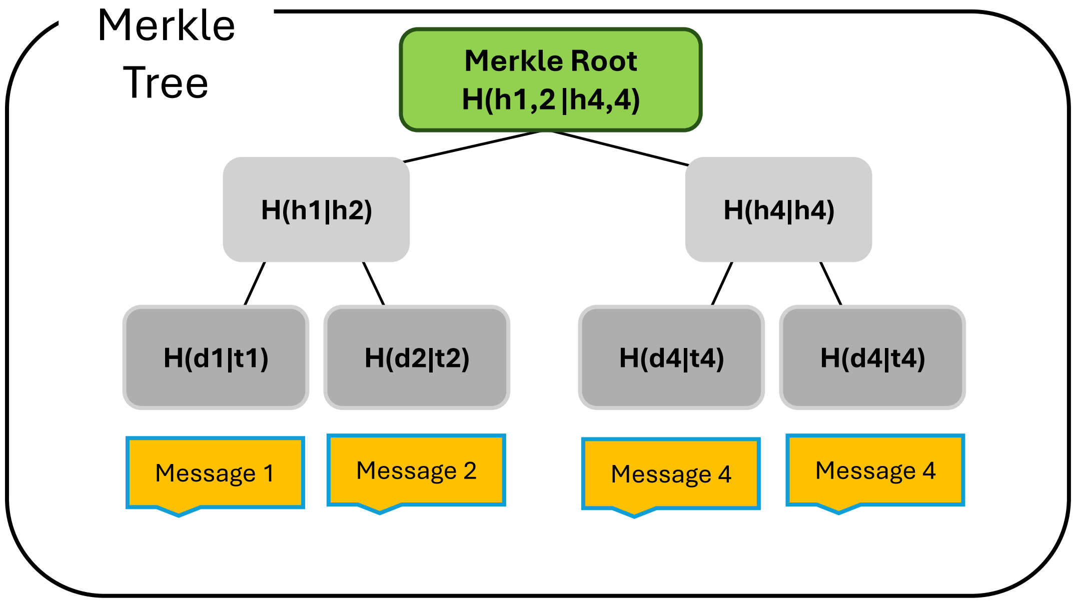
- Use Merkle-tree-based proofs to detect malicious hub behavior
- Key Features:
  - Client-driven: No modification to the hub, no new trusted servers required.
  - Minimal communication overhead.
  - No change to MIMI or MLS.
- High-Level Flow:
  - Clients maintain ordered list of messages and timestamps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

Handling Deleted Messages in Merkle Tree

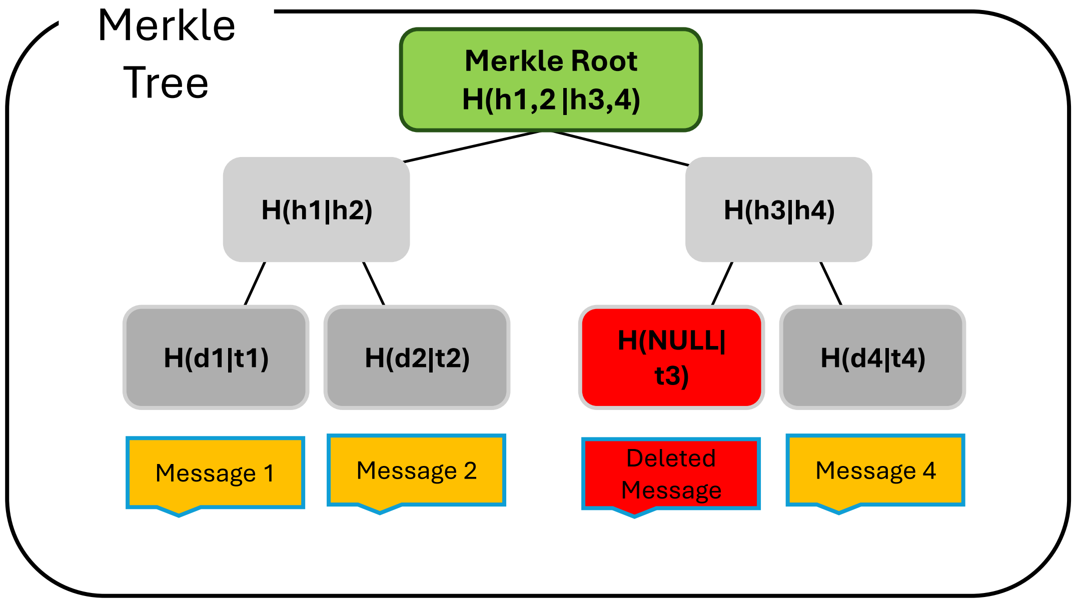


- We propose Tombstones to handle deleted messages in Merkle trees.
  - Tombstones are null placeholders, the original message is replaced with a null value.
- Preserves auditability and verifiability by not modifying the message list indices.

Handling Deleted Messages in Merkle Tree



Merkle Proof Generation without Tombstone

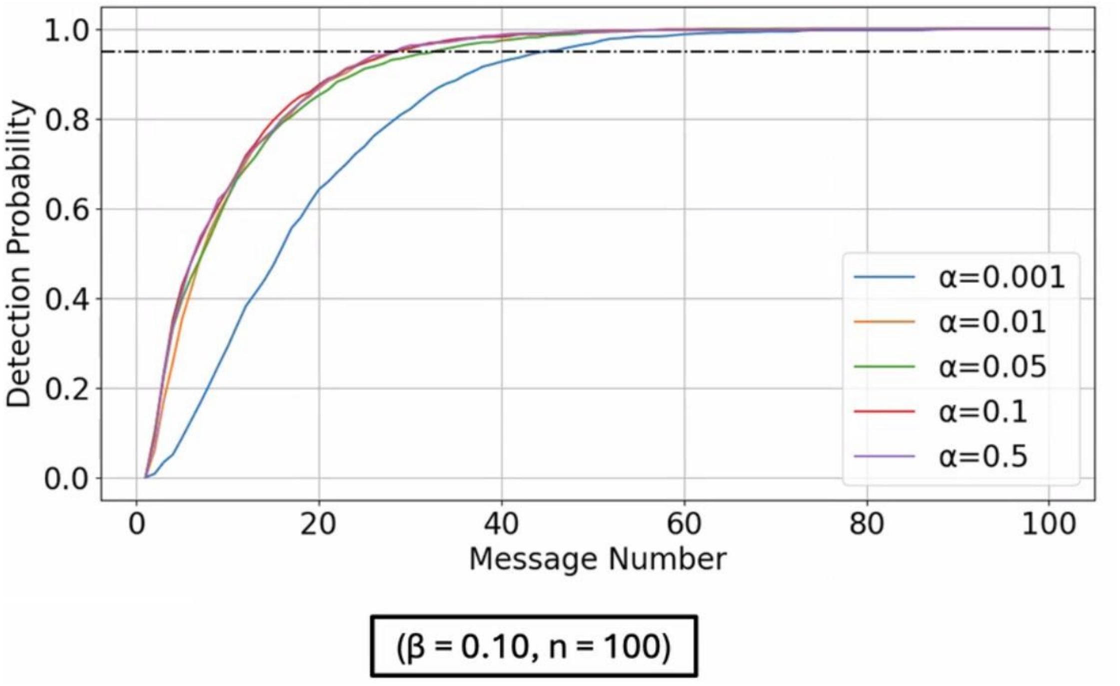


Merkle Proof Generation with Tombstone

Discussion

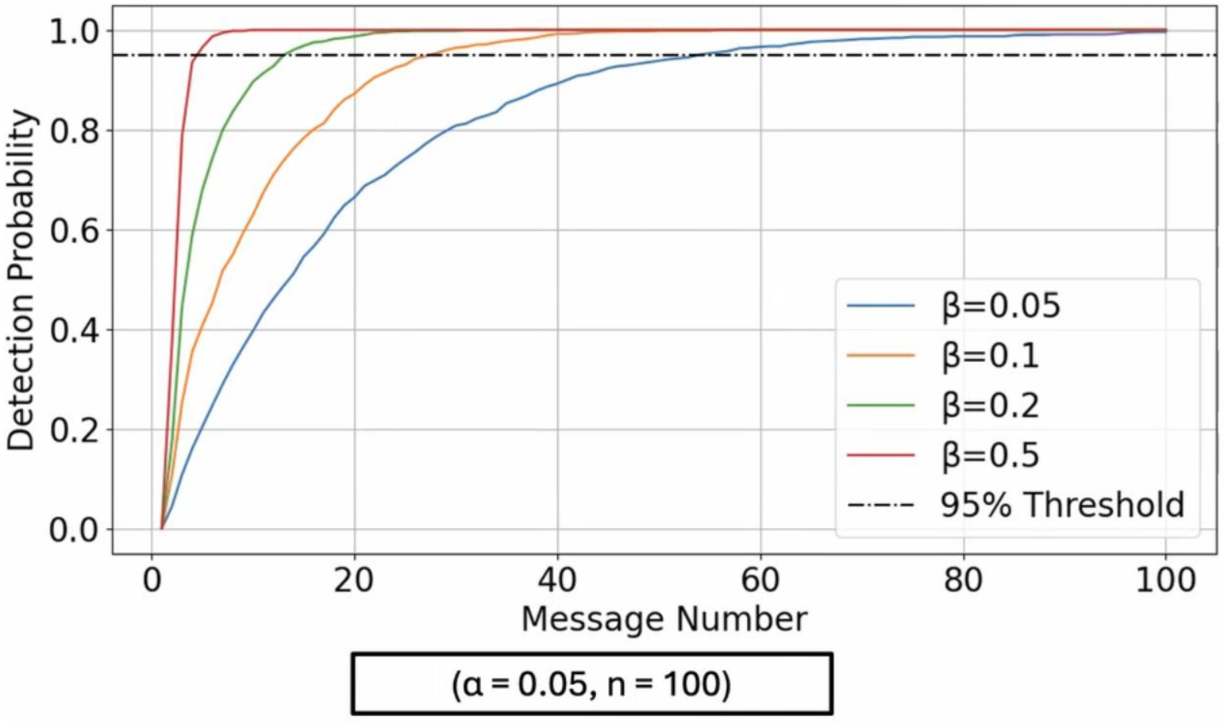
Backup Slides

Impact of Client Sampling Rate α



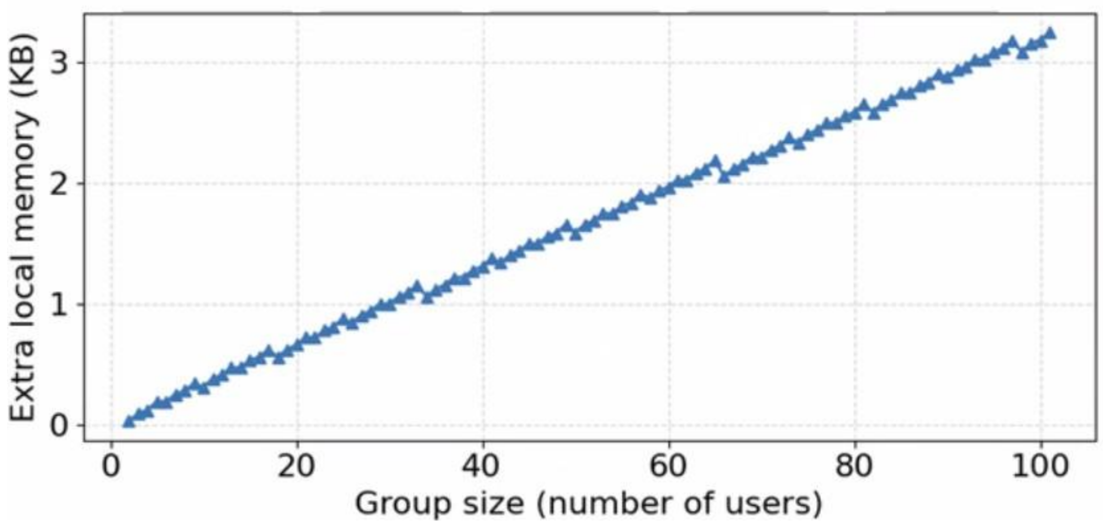
- Performed evaluation using 1000 trial Monte-Carlo simulations with varying group sizes till 100 nodes.
- **Low a (0.001): H**igh detection latency (> 60 messages for 95% detection).
- **Moderate/High a (0.01 - 0.05): S**ignificant improvement in detection speed (optimal point).
- **High a (>0.05):**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

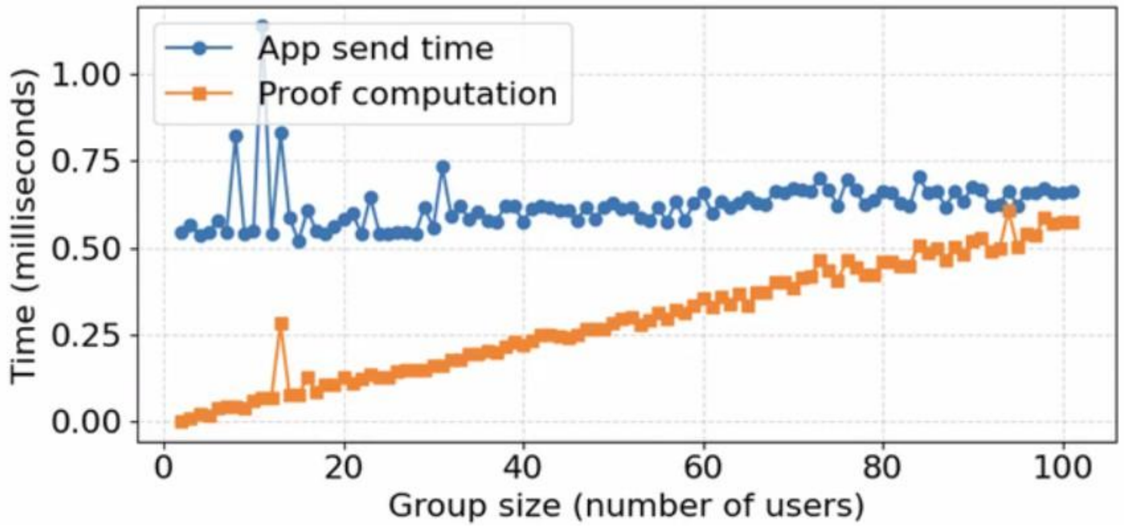
Impact of Client Sampling Rate β



- **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