Detecting presence Of Malicious Hub in MIMI Protocol for Cross-Platform Messaging Interoperability

Harditya Sarvaiya

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Eric Burger, Chair
Melissa Cameron
Na Meng

June 20, 2025
Arlington, Virginia

Keywords: MIMI, MLS, Merkel Tree
Copyright 2025, Harditya Sarvaiya

Detecting presence Of Malicious Hub in MIMI Protocol for Cross-Platform Messaging Interoperability

Harditya Sarvaiya

(ABSTRACT)

The IETF *More Instant Messaging Interoperability* (MIMI) protocol enables *interoperable* group messaging across otherwise isolated services such as WhatsApp, Signal, and Telegram. It routes every Messaging Layer Security (MLS) ciphertext through a central hub that timestamps the message and broadcasts it to all group participants. If the hub is compromised, it can silently drop, delay, or reorder messages, undermining order integrity while leaving end-to-end encryption intact We introduce a lightweight, Merkle-tree-based audit layer that allows clients to detect such misbehavior. Each client stores every received message together with its hub-assigned timestamp in an ordered list. Clients periodically generate a Merkle proof from this list and broadcast it by embedding the proof in an encrypted application message. Because the hub cannot predict which messages carry proofs, it cannot selectively discard them. Upon receiving a proof, other clients verify it and broadcast their own proofs. Any inconsistency is then propagated to the entire room, creating a non-repudiable record of hub misconduct. A Rust prototype built on OpenMLS was evaluated on a 100-node emulated network. With a client sampling rate of 5%, and a hub attack probability of 10%, the scheme detected 95% of message-drop or reordering attacks within the first 40 messages, consumed only 3 kB of additional memory per client, and required less than 1 ms of client-side processing per proof. The audit's memory requirement grows linearly with room size and requires no changes to the hub protocol, providing a practical, low-overhead path to verifiable message-order integrity in large interoperable messaging systems.

Detecting presence Of Malicious Hub in MIMI Protocol for Cross-Platform Messaging Interoperability

Harditya Sarvaiya

(GENERAL AUDIENCE ABSTRACT)

Today's messaging apps like WhatsApp, Signal, and Telegram don't talk to each other, forcing users to stick to a single platform. The new MIMI protocol, developed by the IETF, is changing that by enabling secure group chats across these different services. But there's a catch: MIMI relies on a central hub to forward messages to group members. If that hub is compromised, it can quietly drop, delay, or rearrange messages—even though end-to-end encryption keeps the content safe. Our work tackles this problem by adding a lightweight audit system that helps users detect if the hub misbehaves. Each person in a group keeps a private, tamper-evident log of messages and timestamps using a data structure called a Merkle tree. Occasionally, users send a short encrypted "proof" that reflects their message history. Since these proofs are sent at random times and hidden inside regular messages, a malicious hub can't predict or block them without getting caught. If anyone detects a mismatch, they alert the group, exposing the problem. We tested our approach in a simulated network of 100 users. Even when the hub was acting maliciously 10% of the time, our system detected most problems quickly—with almost no impact on speed or memory use. This makes it a practical tool for improving trust in cross-platform messaging, especially in high-stakes settings like emergency response where reliable communication is critical.

Dedication

Dedicated to my parents and my late maternal grandfather.

Acknowledgments

I would like to express my deep gratitude to Dr. Eric Burger for chairing my thesis committee and mentoring me as a research student. His leadership and guidance has fostered my development as a researcher beyond what I believed achievable. I am also immensely grateful to Dr. Meng and Dr. Cameron for serving on my committee and providing invaluable research inputs. I extend my heartfelt appreciation to my lab mates at CCI for creating such a collaborative and supportive research environment. Their willingness to share knowledge and engaging discussions have enriched my graduate experience immeasurably. Finally, I would like to thank my parents for always believing in me and encouraging me on this journey.

Contents

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Background 1

1.2 Motivation 2

1.3 Contributions 3

1.4 Thesis Organization 4

2 Background 5

2.1 End-to-End Encryption (E2EE) 5

2.1.1 Symmetric-Key Encryption 6

2.1.2 Asymmetric-Key Encryption 7

2.1.3 Key Schedule (Ratcheting) 9

2.2 Security Properties in Group Communication 10

2.3 Regulatory Push for Interoperability 12

2.4 Merkle Trees and Tamper-Evident Logs 13

2.5 Probabilistic Sampling and Monte-Carlo Analysis 15

2.5.1 Probabilistic Sampling 15
2.5.2 Monte-Carlo Analysis 15

3 Review of Related Literature **17**

3.1 End-to-end–encryption protocols 17
3.2 Continuous Group-Key Agreement (CGKA) Protocols 18
3.3 Message-Order Consistency and Threat Models 19
3.4 Merkle-Tree Integrity Mechanisms 20

4 System & Threat Model **23**

4.1 MIMI Architecture 23
4.1.1 MIMI Components 23
4.1.2 Message Flow 25
4.2 Threat Model — Malicious Hub 26
4.2.1 Reordering Attack 26
4.2.2 Sequence-Diagram View 27
4.2.3 Adversary Capabilities 28
4.2.4 Security Goals 28

5 Protocol Design **30**

5.1 Design Requirements and Rationale 30
5.2 Timestamp-Indexed Merkle-Tree Audit Log 31

5.2.1 Leaf Computation 31
5.2.2 Building the Merkle Root 32
5.2.3 Merkle Proof Structure 32
5.3 Probabilistic Proof-Exchange Schedule 33
5.3.1 Sampling Mechanism 33
5.3.2 Indistinguishability 34
5.3.3 Detection Probability 34
5.3.4 Sampling Rate Selection 35
5.4 Client-Side Verification Process 35
5.5 Conditions for Detecting a Malicious Hub 36
5.6 Implementation Overview 37

6 Results **39**

6.1 Detection Probability 39
6.2 Performance Overhead 42
6.3 Scalability Analysis 44
6.4 Discussion of Findings 44

7 Conclusions & Future Work **46**

7.1 Key Findings and Contributions 46
7.2 Limitations 47

7.3 Future Research Directions 47

7.3.1 Adaptive Malicious-Hub Defence 47

7.3.2 Automatic Group Reconciliation 48

7.3.3 Public-Safety Networks 48

7.3.4 Privacy-Preserving User Discovery 48

7.4 Concluding Remarks 49

Bibliography **50**

List of Figures

2.1 Typical symmetric-key encryption workflow. 7

2.2 Public-key encryption from Bob to Alice. 9

2.3 Iterative KDF-chain ratchet: each invocation replaces the current chain key with a successor and yields a one-time output key. 10

4.1 High-level MIMI architecture with one hub and two follower providers. . . . 24

4.2 End-to-end message-delivery flow in MIMI. 25

4.3 A malicious hub delays Client A's order, causing Client B to view a contradictory sequence. End-to-end encryption holds, yet the conversational context is corrupted. 26

4.4 Message-reordering attack in a three-participant room. The hub (Server A) delivers $\mathsf{M_C}$ to Client A first, then $\mathsf{M_B}$, while followers see the opposite order. 27

6.1 Detection probability for a single reordering attack ($\alpha = 0.05$, $\beta = 0.10$, $n = 100$). 40

6.2 Impact of hub aggressiveness β ($\alpha = 0.05$, $n = 100$). 41

6.3 Effect of client sampling rate α ($\beta = 0.10$, $n = 100$). 41

6.4 Detection probability vs. room size ($\alpha = 0.05$, $\beta = 0.10$). 42

6.5 Extra client memory per group size. 43

6.6 Median MLS send latency and proof-generation time. 43

List of Tables

2.1 Selected real-world systems that rely on Merkle-tree proofs. 14

3.1 Representative two-party E2EE protocols 18

3.2 Representative CGKA schemes for scalable E2EE group messaging. 19

KDF DH

List of Abbreviations

ART Asynchronous Ratcheting Tree – A tree-based group-key structure that updates in $\mathcal{O}(\log n)$ time.

CGKA Continuous Group Key Agreement – Protocol class that maintains a shared secret across dynamic group membership.

E2EE End-to-End Encryption – Cryptographic protection ensuring that only endpoints can decrypt the content of messages.

FS Forward Secrecy – A guarantee that past session keys remain secure even if a long-term key is later compromised.

HTTP Hypertext Transfer Protocol – The application-layer protocol used to transport packets messages over TLS.

MIMI More Instant Messaging Interoperability – The IETF protocol suite enabling secure, interoperable group messaging.

MLS Messaging Layer Security – The IETF standard that provides scalable, authenticated, end-to-end encrypted group messaging.

PCS Post-Compromise Security – The ability of a protocol to recover security after an endpoint compromise.

TLS Transport Layer Security – Cryptographic protocol securing communications over computer networks.

TreeKEM Tree Key Encapsulation Mechanism – The MLS key-update algorithm derived from ART.

Chapter 1

Introduction

1.1 Background

The rapid evolution of messaging applications has fundamentally changed how people communicate daily. Despite the widespread adoption of end-to-end encryption (E2EE), most messaging platforms—Signal, WhatsApp, Telegram, iMessage, and others—still operate as isolated silos. In practice, users must install and maintain the same app as their contacts to exchange secure messages, which fragments the communication landscape and limits user choice.

Such fragmentation is increasingly unsustainable. Regulators, particularly in the European Union, have recognized that requiring users to stay within a single application stifles competition and innovation. The EU's new Digital Markets Act (DMA) now forces major messaging providers to interoperate at a basic level, prompting the IETF to form the *More Instant Messaging Interoperability* (MIMI) working group. MIMI aims to define a common, secure messaging standard that spans multiple services, leveraging the Messaging Layer Security (MLS) framework for its strong guarantees of forward secrecy and post-compromise security.

MIMI's draft specification adopts a centralized hub architecture: each MLS-encrypted message is routed through a central hub operated by one service provider. The hub assigns a monotonically increasing timestamp to each message before broadcasting it to all group

participants. Although the hub cannot see plaintext message, it remains fully in charge of message ordering and delivery timing. A malicious or compromised hub could drop, delay, or reorder messages without breaking encryption—undermining the integrity of the group transcript. This vulnerability poses serious risks in use cases where message order is critical, such as emergency response, financial trading alerts, and diplomatic channels.

1.2 Motivation

End-to-end encryption protects message content, but it does not enforce correct or timely delivery. In an interoperable environment where users on different platforms rely on a single hub for message routing, this limitation becomes acute. By design, MIMI's hub stamps each message with a timestamp and broadcasts it, yet nothing prevents the hub from silently reordering or withholding certain messages. In casual conversations, minor delays or reordering might go unnoticed. However, in high-stakes scenarios—dispatch operators coordinating first responders, traders executing time-sensitive orders, or negotiators in diplomatic talks—message-order integrity can be a matter of life, safety, or millions of dollars.

MIMI's architecture implicitly trusts the hub to behave honestly, but in real-world this trust can be undermined. Existing approaches to detect such misbehavior often rely on global transparency logs or federated consensus, neither of which is practical at a global scale. A lightweight lightweight, client-centric mechanism is needed that allows honest participants to catch a misbehaving hub without requiring additional trusted infrastructure.

This thesis is motivated by the need to close this integrity gap. We aim to preserve the benefits of cross-platform, E2EE messaging while empowering clients to verify that the hub has neither dropped nor reordered messages. The proposed solution must adhere to the computational, memory, and bandwidth limitations of mobile based client devices to ensure

broad implementation feasibility while maintaining optimal user experience.

1.3 Contributions

Our main contributions are as follows:

1. **Threat Model.** We rigorously define the attack surface introduced by MIMI's timestamping hub. We show how a malicious or compromised hub can induce message-order inconsistencies—dropping, delaying, or reordering ciphertexts—in a way that remains undetectable without additional cryptographic checks.

2. **Merkle-Tree Audit Based Protocol.** We design a lightweight, timestamp-indexed Merkle tree audit that enables clients to generate compact, tamper-evident proofs of their local message history. By embedding these proofs inside encrypted MLS application messages with small probability α, clients ensure that a malicious hub cannot predict or selectively suppress proofs. Our analysis shows that, with a 10% client sampling rate ($\alpha = 0.10$), single-message tampering is detected with probability 95% within 40 subsequent messages, at the cost of only 3 kB of additional memory per client.

3. **Prototype Implementation.** We implement the proposed audit layer in Rust on top of the OpenMLS library. The prototype seamlessly integrates with standard MLS message flows, requiring no changes to the hub code. Empirical measurements on a 100-node emulated network has around 1ms additional proof-processing time and memory grows linearly with room size.

4. **Evaluation.** Through Monte Carlo simulations and real-time emulation, we evaluate detection accuracy, computational overhead and memory usage impact across varying

group sizes, sampling rates, and attack probabilities. Our results provide practical parameter recommendations for different application contexts—casual social chats, moderately sized workgroups, and mission-critical, high-assurance environments.

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

- **Chapter 2** provides foundational material on the cryptographic and protocol primitives relevant to our design.
- **Chapter 3** presents a systematic review of prior literature, grouping contributions by protocol type: E2EE, CGKA, message-order consistency, and Merkle-based integrity. We identify key formal analyses and recurring gaps that motivate our approach.
- **Chapter 4** details the MIMI protocol architecture, including our system model and defines the threat model.
- **Chapter 5** introduces our core innovation: a timestamp-indexed Merkle audit layer. We describe client-side tree maintenance, probabilistic proof sampling, and the procedures for detecting and resolving hub misbehavior.
- **Chapter 6** presents our empirical findings: detection accuracy, bandwidth and CPU overhead, scalability under adversarial strategies, and results of ablation studies.
- **Chapter 7** restates our contributions, discusses their implications for secure and interoperable messaging in regulated ecosystems, and sketches future directions such as adaptive sampling, emergency communication extensions, privacy-respecting user discovery, and post-quantum resilience.

Chapter 2

Background

This chapter establishes the cryptographic and systems vocabulary used throughout the thesis. It reviews the building blocks of modern end-to-end encryption, the essential security properties for secure group messaging, the policy pressure behind interoperability, and the hash-based audit and analysis techniques we will later extend.

2.1 End-to-End Encryption (E2EE)

End-to-end encryption (E2EE) is the defining security primitive of modern messaging systems: data are transformed into ciphertext on the sender's device and are only ever decrypted on the recipient's device. Every intermediary—including access points, relays, cloud backups, and even the service provider's own servers—handles only opaque octet streams and can neither read nor modify protected content without detection [24, 28]. In practice, an E2EE protocol must include three cryptographic primitives:

1. *Symmetric key encryption* for high-speed bulk encryption.

2. *Asymmetric key encryption* for bootstrapping shared secrets and authenticating devices.[1]

[1]Some protocols, e.g. HPKE, blur this distinction by wrapping symmetric keys inside an asymmetric-key encapsulation.

3. A *state-management logic* (*ratchet* or *key schedule*) that evolves keys across multiple messages and devices.

The subsections below detail each layer and show how they assemble into protocols such as Signal and MLS.

2.1.1 Symmetric-Key Encryption

Symmetric-key encryption employs the same secret key for both encryption and decryption. Its main benefit is speed: once a shared key is in place, even constrained devices can encrypt or decrypt large volumes of data with minimal latency. Accordingly, symmetric ciphers are the default choice for (i) bulk data protection in end-to-end encrypted messaging, (ii) file and disk encryption at rest, and (iii) performance-sensitive network protocols such as TLS 1.3 [42] where asymmetric primitives are only used for initial key-exchange handshake. Figure 2.1 illustrates encryption from Bob to Alice using a single shared secret key. In a symmetric-key setting the green and red keys shown here are *identical*—both Alice and Bob hold the same secret and use it for encryption and decryption.

Cipher Suites and AEAD Constructions. Symmetric-key primitives—specifically block ciphers such as AES[37] and Camellia[34], as well as stream ciphers exemplified by ChaCha20[39]—constitute the primary mechanisms for high speed end-to-end encryption. These algorithms are capable of achieving gigabit-per-second throughput. In order to provide simultaneous confidentiality and integrity guarantees, modern protocols uniformly employ an authenticated-encryption-with-associated-data (AEAD) paradigm such as AES-GCM[37] and ChaCha20-Poly1305[39], which ensure that any attempt at ciphertext manipulation is detected upon decryption [35].

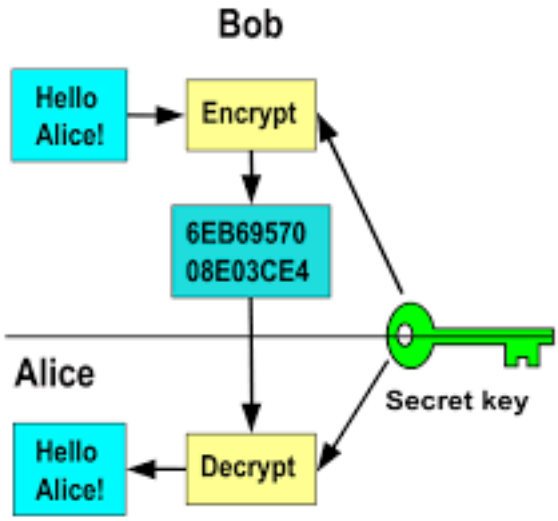

Figure 2.1: Typical symmetric-key encryption workflow.

Key-Distribution Challenges. The primary limitation of symmetric-key encryption lies in its reliance on a single, pre-shared secret: each recipient must possess the correct key *prior* to decrypting any given message. As the number of communication participants, devices, or concurrent sessions increases, the overhead associated with secure key distribution and rotational management increases substantially [26]. Consequently, group messaging systems typically integrate symmetric encryption with asymmetric key-establishment protocols (see Section 2.1.2), thereby enabling the derivation of fresh session keys on demand while preserving AEAD protections for data both at rest and in transit.

2.1.2 Asymmetric-Key Encryption

Public-key cryptography mitigates the symmetric key-distribution problem by splitting each credential into a widely shared *public* key and a closely guarded *private* key. Because the private component never leaves its secure boundary (e.g., user device), any party can initiate a secure session simply by knowing the recipient's public key. Figure 2.2 depicts this one-way

encapsulation–decapsulation workflow between Alice and Bob. Bob encrypts with Alice's *public* key (green); only Alice can decrypt using her *private* key (red). No pre-shared secret is required.

Key-agreement primitives such as X25519[29] or P-256[38] derive a high-entropy shared secret across an untrusted channel; the resulting secret seeds the AEAD layer described in Section 2.1.1.

Digital-signature schemes like Ed25519[22] and ECDSA[38] provide source authentication and replay protection. In forward-secure group protocols, each epoch rotates its own signature key as part of the ratchet mechanism [9].

Hybrid Constructions and Security Guarantees. Real world protocols rarely encrypt bulk data with public-key primitives directly, owing to their comparatively high latency and bandwidth cost. Instead, hybrid schemes—most notably Hybrid Public-Key Encryption (HPKE)[5] and TLS 1.3[42] use high-throughput AEAD ciphers for encrypting in bulk while using public-key operations only for initial session setup.

Performance Trade-offs. Asymmetric key encryption remains orders of magnitude slower than symmetric encryptions. By confining asymmetric work to the handshake and leveraging tree-structured Diffie–Hellman (as in MLS), group messaging systems gain scalable key establishment without sacrificing high throughput offered by AEAD.

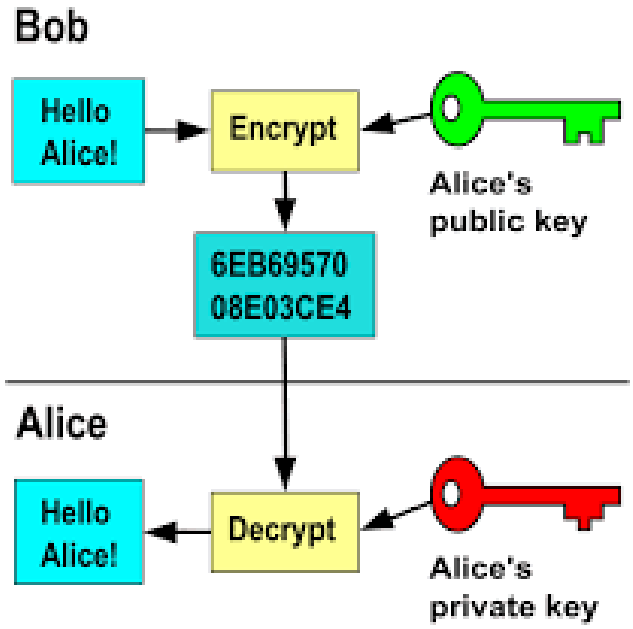

Figure 2.2: Public-key encryption from Bob to Alice.

2.1.3 Key Schedule (Ratcheting)

A *key schedule*—often referred to as a *ratchet*—is a state-management algorithm that evolves cryptographic keys in one direction only, ensuring that each new secret is unlinkable to its predecessors. Two security goals follow immediately:

- **Forward secrecy** Past ciphertexts remain confidential even if a current key is compromised.
- **Post-compromise secrecy** Once an adversary loses access to the device, future keys (and messages) recover their confidentiality without manual re-keying.

KDF-Chain Ratchet. Figure 2.3 shows the *symmetric-ratchet* used by Signal's Double Ratchet [41] and by the MLS key schedule [6]. Each step feeds a *chain key* CK_i and message-

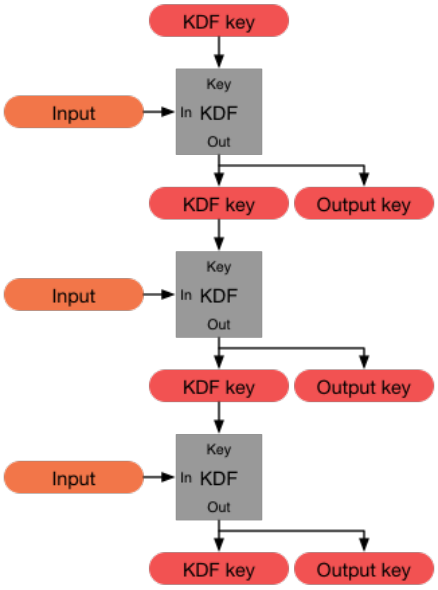

Figure 2.3: Iterative KDF-chain ratchet: each invocation replaces the current chain key with a successor and yields a one-time output key.

specific input into a key-derivation function (KDF) to obtain a fresh output key MK_i and the successor chain key CK_{i+1}. After use, CK_i and MK_i are removed, preventing backward derivations.

Limitations and Open Questions. State-of-the-art protocols still leak *metadata*—participants, traffic volumes, and timing—or defer transcript ordering to a trusted hub [16, 45]. Closing these gaps requires tamper-evident audit layers as discussed in (Chapter 5).

2.2 Security Properties in Group Communication

The security goals of modern group-messaging protocols extend well beyond point-to-point confidentiality. This section lists the core properties that any interoperable, end-to-end-

encrypted (E2EE) system must satisfy.

Confidentiality & Authentication Every application-layer payload is protected with an AEAD construction such as AES-GCM or ChaCha20-Poly1305. AEAD simultaneously encrypts the plaintext and computes an integrity tag over both the ciphertext and selected header fields (e.g., group ID, epoch counter), preventing replay attacks [35]. A message that fails authentication is silently discarded, guaranteeing sender authenticity and message integrity as long as the underlying keys remain secret.

Forward Secrecy Forward Secrecy guarantees that an adversary who later compromises a participant's long-term private keys (e.g., identity or signing keys) is still unable to decrypt any ciphertexts that were captured in the past. The property is realised by periodically deriving fresh session keys—typically via ephemeral Diffie–Hellman exchanges—and securely deleting the old keys once they have served their purpose. Because those ephemeral secrets are never stored long-term, reconstructing past session keys after a compromise becomes mathematically infeasible.

Post-Compromise Security (PCS) Also termed *break-in recovery*, PCS addresses the complementary scenario: the adversary temporarily compromises a device and learns its current secret state. A robust ratchet allows the device to heal once the attacker loses access. Whenever an honest group member publishes a fresh Diffie–Hellman key, the protocol adds that new key into its evolving key schedule, overwriting the old secrets. As a result, any keys the attacker previously stole are immediately rendered useless. Both Signal's Double Ratchet and MLS's tree ratchet achieve PCS while preserving forward secrecy guarantees [6, 9, 41].

Transcript / Message-Order Consistency In a multi-party setting, confidentiality alone is insufficient; participants must agree on what was said and in what order. We re-

quire that all honest members observe a single, total-ordered transcript of application messages. Centralised deployments often rely on a trusted server (or "hub") to assign monotonic sequence numbers or timestamps. But in untrusted servers this introduces a single point of failure: the server can drop, delay, or reorder messages without detection.

2.3 Regulatory Push for Interoperability

Past attempts at cross-platform messaging— most famously XMPP federation and, more recently, ad-hoc gateways that proxy traffic between closed platforms—have either dropped end-to-end encryption (E2EE) or required users to place full trust in the bridging server, defeating the point of E2EE altogether. As a result, mainstream services have remained fragmented into isolated silos.

The EU Digital Markets Act (DMA). Seeking to break this deadlock, the European Union passed the Digital Markets Act—Regulation (EU) 2022 which designates large platforms as gatekeepers and obliges them to offer secure interoperability for "number-independent interpersonal communication services" (Article 7)[1]. Gatekeepers (like Apple, Meta, Alphabet, Microsoft, Amazon, and ByteDance) were required to allow third-party messengers services to request, test, and audit cross-network connectivity. Crucially, Article 7(8) limits the personal data gatekeepers may share, forcing any technical solution to maintain GDPR-level privacy guarantees. These legal constraints elevate E2EE from a nice-to-have to a regulatory requirement.

To provide a vendor-neutral venue for the technical work, the IETF chartered the **More Instant Messaging Interoperability** (MIMI) working group[7].

2.4 Merkle Trees and Tamper-Evident Logs

Merkle trees were introduced by Ralph Merkle in 1979 [36] as an efficient way to apply one-way hash functions to large data sets. Although originally proposed for digital signatures, the structure soon found broader use in file systems, peer-to-peer networks, and, more recently, blockchain and software-supply-chain security. Today a Merkle trees are the central building block for any tamper-evident log: an append-only data structure whose integrity can be verified with logarithmic proofs.

Definition and basic operations. A Merkle tree is a full binary tree in which each internal node stores the hash of its two children, and each leaf stores the hash of an individual record. Let $H\colon \{0,1\}^* \to \{0,1\}^\lambda$ be a collision-resistant hash function. For leaf i with data block d_i,

$$\ell_i \;=\; H(d_i), \qquad v_{\text{parent}} \;=\; H(v_{\text{left}} \,\|\, v_{\text{right}}),$$

where "$\|$" denotes concatenation. The hash of the root node, R, *commits* to every block in the tree.

Membership proof (inclusion proof) To convince a verifier that block d_m is present, the prover supplies the sibling hashes on the unique path from ℓ_m to R. The verifier recomputes the path hashes in $O(\log n)$ time and checks that the resulting root equals the known commitment R.

Append operation A simple append rebuilds the tree, but practical implementations use sparse or 2^k-ary variants to achieve amortised $O(\log n)$ updates [14].

Consistency proof Given two roots R_{old} at size n and R_{new} at size $n' > n$, a *consistency proof* demonstrates that R_{old} is a prefix of the larger tree—crucial for detecting

truncation attacks on append-only logs [31].

Applications. Table 2.1 summarises representative deployments.

Domain	Role of Merkle Tree
Git / Mercurial	Commit history integrity[17]
BitTorrent	Verifiable chunk download[33]
Bitcoin, Ethereum	Block header commitment to transactions
Certificate Transparency	Append-only log of issued TLS certificates[40]
Supply-chain SBOMs	Tamper-evident software bills of materials[27]

Table 2.1: Selected real-world systems that rely on Merkle-tree proofs.

Algorithms and variants. Modern systems optimise the classic construction in three orthogonal directions:

- *Sparse Merkle trees* store only non-default nodes, enabling 2^{256}-size key spaces with $O(\log n)$ storage[30].
- *Vector commitments* (e.g., Append-Only Vector Commitments, AOVCs) add succinct proofs for both inclusion and position, useful in stateless blockchain clients[15].
- *Verkle trees* replace hashes with inner-product commitments for sub-logarithmic proofs and are under evaluation for Ethereum scaling[10].

Security guarantees. Provided the underlying hash function is collision-resistant and pre-image-resistant, an adversary cannot forge an inclusion or consistency proof without being able to find hash collisions. When embedded in a transparency log with message order auditing, the construction gives strong *fork detection*: any attempt to present inconsistent views is eventually exposed.

2.5 Probabilistic Sampling and Monte-Carlo Analysis

2.5.1 Probabilistic Sampling

Probabilistic sampling is a data-collection technique in which each member of a population is selected with a known, non-zero probability. Because the selection process is random, the resulting *sample* constitutes an unbiased—though approximate—representation of the full population.

- **Goal** Estimate population parameters (mean, proportion, error rate, etc.) while measuring only a fraction of all items.
- **Motivation** Complete enumeration may be infeasible owing to cost, time, or physical constraints.
- **Examples**
  1. Opinion polls that survey a few thousand voters to forecast an election.
  2. Quality-control audits that test a small batch of products instead of the entire production run.

2.5.2 Monte-Carlo Analysis

Monte-Carlo analysis is a computational method that evaluates a mathematical or physical model under a large number N of randomly generated input scenarios [18]. The empirical distribution of the outputs approximates the true distribution one would observe across all possible real-world conditions.

- **Procedure**

1. Define the model or system to be studied.
2. Specify probability distributions for each uncertain input parameter.
3. Draw N random input sets and run the model for each one.
4. Aggregate the outputs to compute statistics such as the mean, variance, or tail risk.

- **Applications**

1. Pricing complex financial derivatives.
2. Estimating project completion times with uncertain task durations.
3. Simulating radiation dose distributions in medical physics.

Chapter 3

Review of Related Literature

This chapter surveys prior work in four areas that underpin the thesis: (i) end-to-end–encryption (E2EE) protocols, (ii) continuous group-key-agreement (CGKA) protocols, (iii) message-ordering and threat models, and (iv) Merkle-tree–based integrity mechanisms. Tables summarise the core features of the protocols.

3.1 End-to-end–encryption protocols

Although large-scale group messaging now dominates the feature lists of WhatsApp, Signal, and Telegram, every such system ultimately inherits its cryptographic core from a two-party channel.

Signal / Double Ratchet

Signal combines a one-time key exchange with a Double Ratchet that intertwines a Diffie–Hellman ratchet with a symmetric KDF chain, ensuring that every message key is unique and that secret material advances monotonically [41]. Symbolic proofs in Tamarin and computational analyses in Coq confirm FS and PCS under the classic insider–outsider model [2]. The protocol additionally offers cryptographic deniability: because session keys are derived from ephemeral DH shares that are later deleted, transcripts cannot be publicly

verified.[48]

WhatsApp "Sender Keys" Extension

To reduce Signal's key distribution cost in groups, WhatsApp unicasts a single sender key (a random symmetric key) from each participant to every other once, after which all outbound messages from that sender carry only a MAC [3]. Sender Keys preserves FS for the two-party channel but does not bind the global transcript: a malicious server can re-order ciphertexts without detection, a weakness formalised in recent cryptographic models.[45]

Protocol	Year	Core Idea	Security Properties
Signal / Double Ratchet[2]	2016	Combines DH and KDF ratchets; each message uses a unique key	FS[1], PCS, deniability
Sender Keys (WhatsApp)[3]	2017	One-time unicast symmetric key per sender, then MACs per ciphertext	FS (two-party), no global transcript binding

Table 3.1: Representative two-party E2EE protocols

The Signal Double Ratchet remains the de-facto standard for two-party messaging; subsequent systems mostly tweak transport or metadata hiding while inheriting Signal's cryptographic core. Nevertheless, group scalability requires CGKA techniques.

3.2 Continuous Group-Key Agreement (CGKA) Protocols

Why CGKA? Two-party ratchets do not scale: a room with n members would require $O(n^2)$ pairwise channels, and offline users would slow down the key updates. CGKA proto-

cols solve this by giving the group a single secret that any member can update while others are offline, yet still guaranteeing forward secrecy (FS), post-compromise secrecy (PCS), and sub-linear communication cost. Modern CGKA schemes also provide asynchronous delivery, meaning that ciphertexts remain decryptable even when messages arrive out of order. [13]

Protocol	**Year**	**Key-Update Mechanism**
ART [13]	2018	Balanced DH tree; update touches $O(\log n)$ nodes
TreeKEM [8]	2023	Tree-DH encapsulation + hash ratchet; core of IETF MLS
TreeSync [49]	2023	Authenticated state-update layer over TreeKEM

Table 3.2: Representative CGKA schemes for scalable E2EE group messaging.

3.3 Message-Order Consistency and Threat Models

End-to-end encryption (E2EE) protects payload confidentiality, and forward/post-compromise secrecy ensure key evolution; yet neither property guarantees that all honest participants view the same conversation in the same order. A malicious or faulty delivery server can silently drop, delay, or re-order messages. When messages convey group-membership changes, payment prompts, or emergency instructions, even a single re-ordering can break application semantics. The research community therefore augments classic secrecy goals with a stronger requirement often called message-order integrity [23].

Models operating in adversarial central server environments

Talek Cheng *et al.* present a system that hides access patterns by spreading ciphertexts across an *any-trust* set of servers, where a single honest server suffices for privacy and

integrity, though this approach increases median latency to seconds [11].

Mobile CoWPI Schliep *et al.* develop a protocol requiring each ciphertext to carry matching inclusion proofs from *multiple* providers, where clients accept messages only after all proofs verify, enabling detection of forged views even when all but one server are malicious; their framework implements servers that co-sign monotonic counters or timestamps attached to each ciphertext, providing verifiable ordering guarantees through cryptographic attestation [44].

Hash-chain audit Komo *et al.* propose embedding every WhatsApp-style message in a signed hash chain, ensuring that any deletion or re-ordering breaks the chain and becomes publicly auditable; they demonstrate how clients can link consecutive messages via hashes or build append-only Merkle trees whose root attests to the entire transcript, ensuring tamper-evident message ordering [25].

Mainstream messengers—including WhatsApp, Signal groups, and iMessage—still assign global order via central trusted server timestamps or sequence numbers, leaving users exposed to subtle drop/re-order attacks. Research prototypes either incur high latency (Talek) or require multiple federated providers (CoWPI), both impractical for interoperable messaging. The audit layer proposed in Chapter 5 addresses this gap: it retains a single MIMI hub but equips clients with timestamp-ordered Merkle proofs that expose misbehaviour with minimal overhead.

3.4 Merkle-Tree Integrity Mechanisms

An append-only blockchain secures data by replicating it under consensus, but that consensus incurs high latency and energy cost. Merkle proofs give nearly the same tamper-evidence

with a fraction of the overhead: a single global root hash authenticates an entire data structure, and logarithmic-size proofs convince verifiers of membership, non-membership, or append-only consistency. Modern secure logging, transparency services, and even database ledgers therefore embed Merkle trees as their integrity core [21].

Transparency Logs. Certificate Transparency uses public append-only logs based on a single Merkle tree, but its minute-scale update windows make real-time auditing impractical. Merkle^2 replaces the single tree with a nested two-level design, lowering monitoring and lookup costs to logarithmic time and supporting sub-second update propagation [20].

On-device block-listing. Thomas *et al.* describe a privacy-preserving security service in which clients query harmful-content block lists held by a remote enforcer without revealing their own data. The design combines private set intersection for query privacy, cryptographic signatures for transparency, and Merkle inclusion proofs so that clients can audit list consistency over time [47].

Merkle-CRDTs. Sanjuán *et al.* merge conflict-free replicated data types (CRDTs) with content-addressed Merkle-DAGs to build "Merkle-CRDTs," a framework that provides logical clocks and tamper-evident replication for large, weakly-connected peer-to-peer systems. The approach leverages the de-duplication and security properties of Merkle trees while allowing massive numbers of replicas to converge without a trusted server [43].

Timestamp-ordered Merkle trees. Sutton and Samavi target integrity guarantees for evolving RDF data. Their scheme inserts each triple into a Merkle tree keyed by an intrinsic timestamp extracted from the data itself; the sorted layout enables $\mathcal{O}(\log n)$ membership and *non-membership* proofs while supporting efficient appends as the

dataset grows. Experimental results show lower update cost and stronger resistance to tampering than prior hash-chain approaches [46].

Prior work shows that Merkle roots can authenticate both who may speak (certificate or membership logs) and in what order they speak (message-hash chains, timestamp trees). Chapter 5 leverages these insights by embedding a timestamp-ordered Merkle audit layer inside MIMI, while our audit proofs expose hub misbehaviour with minimal bandwidth overhead.

Chapter 4

System & Threat Model

4.1 MIMI Architecture

The More Instant Messaging Interoperability (MIMI) protocol elevates the cryptographic assurances of Messaging Layer Security (MLS) into an inter-provider setting, enabling secure, end-to-end–encrypted group chats that span distinct messaging providers. MIMI introduces two server roles Hub and Follower and organises all interaction around the logical notion of room. Because MLS already provides confidentiality, authenticity, forward secrecy, and post-compromise secrecy, the main challenge is to carry those guarantees across provider boundaries [4, 7].

4.1.1 MIMI Components

Figure 4.1 sketches the four principal actors.

- **Room.** A virtual space that holds (i) an MLS group context, (ii) the participant list, and (iii) a role-based authorisation policy.
- **Hub Provider.** The central server for a room. It timestamps every incoming message, fans the message out to followers, and sends back delivery reports. A malicious hub could assign spurious timestamps or omit/delay messages.

- **Follower Providers.** Providers that host users in rooms where another provider acts as hub. Followers forward client messages to the hub and deliver hub messages to local clients.

- **Clients.** End-user devices that maintain long-term identity keys and participate in MLS groups; they rely on MLS for cryptographic protection.

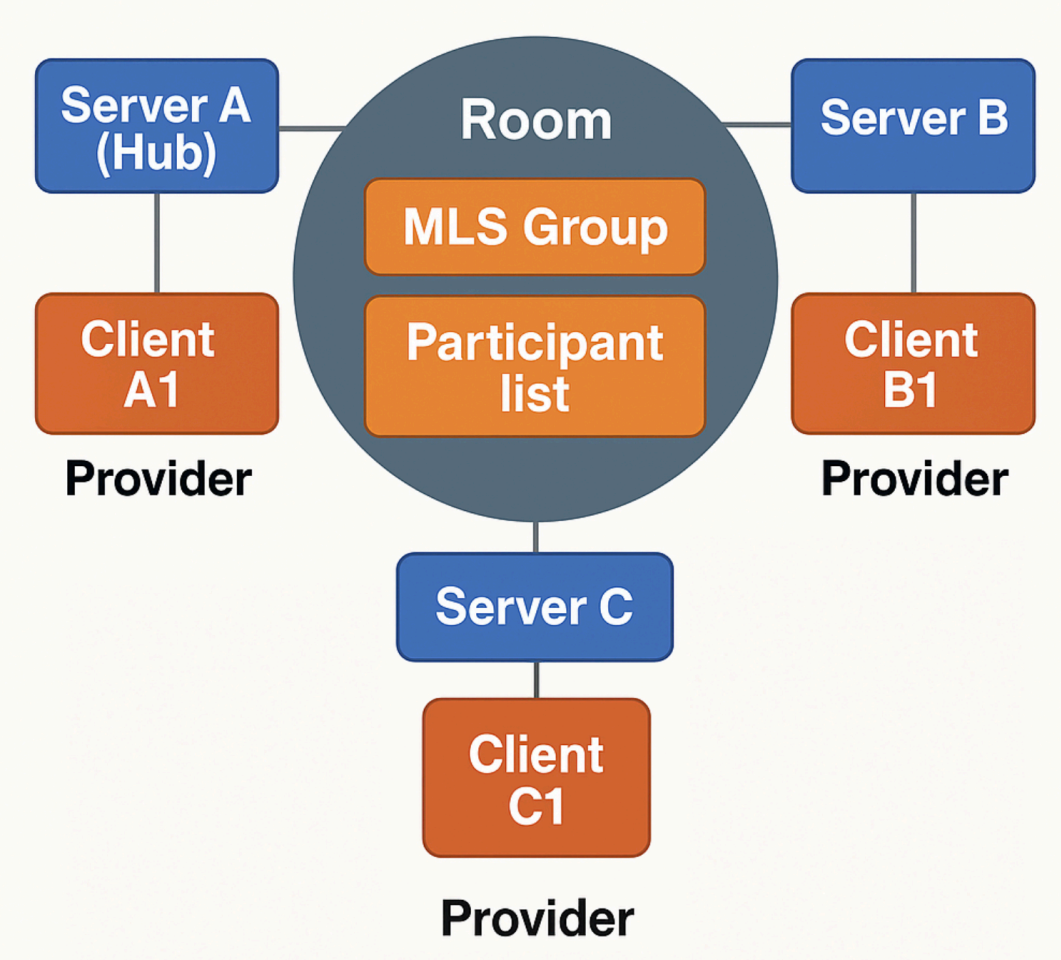

Figure 4.1: High-level MIMI architecture with one hub and two follower providers.

4.1.2 Message Flow

The end-to-end delivery path follows the sequence Client → Follower → Hub → Follower(s) → Client(s), as illustrated in Figure 4.2.

1. **Encrypt.** Client C_1 creates an MLS `PrivateMessage` and sends it to its local follower Server_C.

2. **Timestamp.** Server_C forwards the message to the hub (Server_A), which assigns a monotonically increasing timestamp.

3. **Fan-out.** The hub broadcasts the message (plus timestamp) to every follower and returns a delivery report to Server_C.

4. **Delivery.** Each follower hands the message to its local clients; Server_C also passes the delivery report back to the originating client.

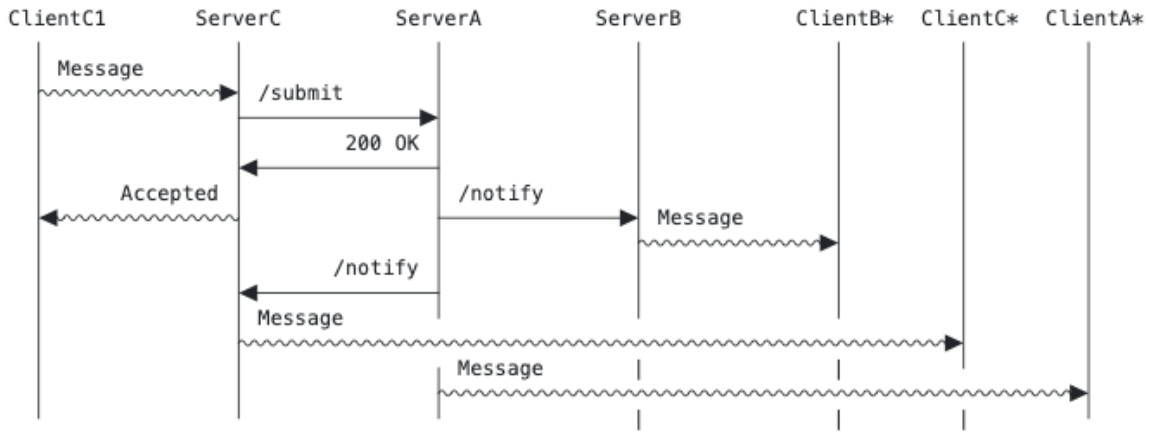

Figure 4.2: End-to-end message-delivery flow in MIMI.

4.2 Threat Model — Malicious Hub

The MIMI draft assigns one privileged task to the *hub*: timestamp every message in arrival order and distribute it to all followers. A compromised or malicious hub therefore becomes a single point of failure. Three types of attack scenarios are possible:

- **Timestamp authority.** The hub may back-date, skip, or duplicate timestamps, creating a false message history.
- **Reordering via delay.** Selectively buffering message delivery lets the hub reorder honest messages at will.
- **Isolated views.** Because clients hear only from the hub, they cannot cross-check their transcript with other participants or an external log.

4.2.1 Reordering Attack

Seen by Client B	Seen by Client A (delay)
09:00 Client C: Lunch ideas?	**09:00** Client C: Lunch ideas?
09:01 Client A: Ordering *pepperoni* pizza!	**09:02** Client B: No pepperoni, veggie please.
09:02 Client B: No pepperoni, veggie please.	**09:03** Client A: Ordering *pepperoni* pizza!

Figure 4.3: A malicious hub delays Client A's order, causing Client B to view a contradictory sequence. End-to-end encryption holds, yet the conversational context is corrupted.

Figure 4.3 shows how a single two-second delay can invert message ordering for the clients: Client B's dietary request arrives *before* Client A's pizza order locally, but *after* it for Client A.

4.2.2 Sequence-Diagram View

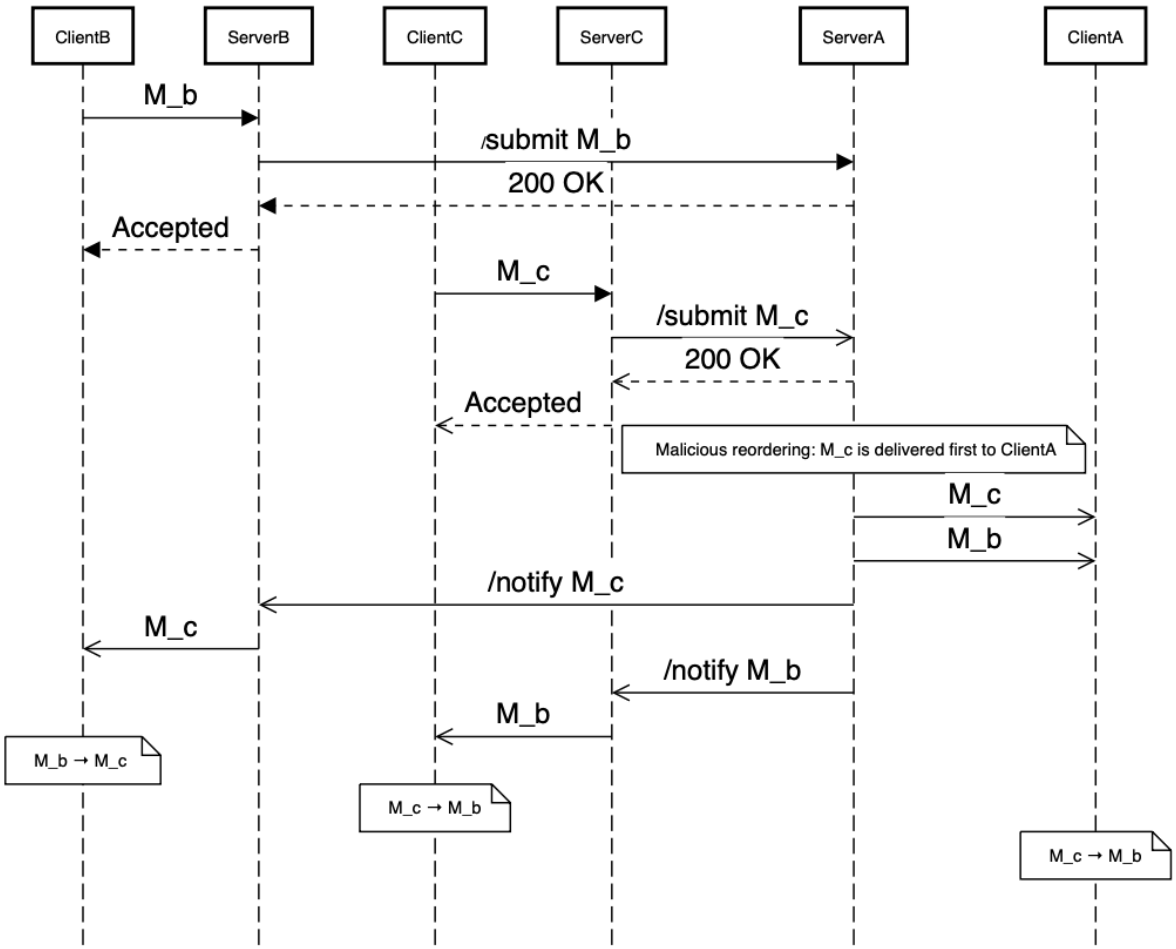

Figure 4.4: Message-reordering attack in a three-participant room. The hub (Server A) delivers $\mathsf{M_C}$ to Client A first, then $\mathsf{M_B}$, while followers see the opposite order.

Figure 4.4 depicts the same attack as a sequence diagram:

1. **Submit $\mathsf{M_B}$.** Client B encrypts $\mathsf{M_B}$ and submits it through Server B. The hub replies `200 OK`.

2. **Submit $\mathsf{M_C}$.** Shortly after, Client C submits $\mathsf{M_C}$ via Server C; again `200 OK`.

3. **Malicious step.** Instead of forwarding in arrival order, the compromised hub timestamps and delivers $\mathsf{M_C}$ *first* to Client A, then sends $\mathsf{M_B}$. Followers receive notifications

in the original order, leaving Clients A and B with divergent transcripts.

The dashed "transcript" boxes at the bottom show the inconsistency: $\langle \mathsf{M_B}, \mathsf{M_C} \rangle$ for Clients B versus $\langle \mathsf{M_C}, \mathsf{M_B} \rangle$ for Client A and C. MLS still protects confidentiality and authenticity, yet order integrity is silently broken—highlighting the need for the Merkle-tree audit layer developed in Chapter 5.

4.2.3 Adversary Capabilities

- **Delivery control.** Arbitrary delay, drop, duplication, or reordering of any ciphertext that crosses the hub.
- **Log tampering.** Forged delivery reports and manipulated timestamp sequences.
- **Cryptographic limits.** The adversary *cannot* break MLS confidentiality or authenticity; private keys remain safe.

4.2.4 Security Goals

The Merkle-audit layer (Chapter 5) aims to provide:

1. **Timely detection.** Dropping, delaying, or reordering any single message is detected with high probability within k subsequent messages.
2. **Non-repudiable evidence.** Each alert includes a verifiable Merkle proof so third parties can confirm misconduct.
3. **Liveness.** When the hub behaves, the audit layer introduces negligible bandwidth, CPU, and memory overhead.

The next chapter details the concrete protocol that realises these goals and the evaluation chapter quantifies its effectiveness.

Chapter 5

Protocol Design

This chapter presents our lightweight Merkle-tree–based audit protocol that augments MIMI's existing MLS flows to detect malicious hub behavior. We begin by stating the design requirements, then detail each component: constructing a timestamp-indexed Merkle tree, probabilistic proof sampling, embedding proofs in encrypted MLS messages, and client-side verification. We conclude with implementation details.

5.1 Design Requirements and Rationale

Our audit layer must satisfy the following properties:

1. **No Additional Trusted Servers.** Clients alone detect hub misbehavior; we do not introduce new infrastructure that must be trusted.

2. **Zero Hub Modifications.** The hub's code and protocol flows remain unchanged.

3. **High Detection Probability.** Single-message drops, delays, or reorders should be detected with probability $\geq 95\%$ within a bounded number of subsequent messages.

4. **Minimal Overhead.** Client-side computation per proof must be sub-millisecond even for rooms of size 100+, and additional memory should remain on the order of kilobytes per client.

5. **Scalability.** The mechanism must work for small social groups ($n \approx 5$) as well as large "channel" settings ($n \approx 100$), with detection latency and overhead that scale gracefully.

To meet these requirements, we leverage a Merkle tree whose leaves are formed by hashing the concatenation of each plaintect message and its hub-assigned timestamp. By sampling proof broadcasts randomly—and encapsulating proofs within normal MLS messages—clients ensure that the hub cannot predict or selectively suppress audit messages.

5.2 Timestamp-Indexed Merkle-Tree Audit Log

Each client C maintains locally an ordered list of all messages it has received, along with the corresponding hub-assigned timestamps. Concretely, let

$$M = [(d_1, t_1), (d_2, t_2), \ldots, (d_m, t_m)]$$

denote the client's local transcript, where d_i is the plaintext message payload of the ith message and t_i is the timestamp attached by the hub. We assume $t_1 < t_2 < \cdots < t_m$ under normal (honest) operation.

5.2.1 Leaf Computation

For any contiguous subrange of messages $i = s, \ldots, e$, define each leaf hash

$$\ell_i \;=\; H(d_i \,\|\, t_i),$$

where H = SHA-256 (in our implementation but any other collision-resistant hash can be used) and $\|$ denotes byte-concatenation. These leaves are sorted by the timestamp values t_i.

5.2.2 Building the Merkle Root

To compute the Merkle root $R_{s,e}$ over messages s through e, we:

1. Collect leaves ℓ_s, ℓ_{s+1}, $\ldots, \ell_e$.
2. If the number of leaves $(e - s + 1)$ is not a power of two, duplicate the last leaf until reaching the next power of two.
3. Pairwise hash adjacent leaves to form parent nodes:

$$n_j = H\big(\ell_{2j-1} \,\|\, \ell_{2j}\big), \quad j = 1, 2, \ldots$$

4. Recursively hash siblings until a single root $R_{s,e}$ remains.

Each client stores only the current Merkle tree of all received messages (i.e., $s = 1$, $e = m$), but can re-derive roots for any subinterval $[s, e]$ when generating a proof.

5.2.3 Merkle Proof Structure

A proof for the subrange $[s, e]$ consists of the tuple

$$\big(R_{s,e},\, t_{\max},\, s,\, e\big)$$

where

- $R_{s,e}$ is the Merkle root of leaves $\ell_s, \ldots, \ell_e$.

- $t_{\max} = \max\{\, t_i : i = s, \ldots, e\}$ is the largest timestamp in that interval.

5.3 Probabilistic Proof-Exchange Schedule

If we broadcast a proof for $[s, e]$ on every message, the hub could block all such "audit" messages without detection. Instead, we use randomized sampling driven by a parameter $\alpha \in (0, 1)$.

5.3.1 Sampling Mechanism

Each time client C receives a new message (i.e., the mth ciphertext), it performs:

$$\text{flip} \sim \text{Bernoulli}(\alpha).$$

If flip $= 1$, then:

Client generates a proof for $[1, m]$ (or for a sliding window).

Otherwise, no proof is sent.

We denote this sub-procedure:

Algorithm 1 SampleAndBroadcastProof(α)

1: Receive new ciphertext (d_m, t_m).
2: Append (d_m, t_m) to local list M.
3: **if** Bernoulli(α) == 1 **then**
4: $\big(R, t_{\max}, s, e\big) \leftarrow$ ComputeMerkleProof(M, s, e)
5: Encapsulate $\big(R, t_{\max}, s, e\big)$ inside an encrypted MLS `PrivateMessage`.
6: Broadcast this MLS message to the hub for fan-out.
7: **end if**

Here, ComputeMerkleProof(M, s, e) returns a valid proof for messages s through e. One can choose $s = 1$ (full transcript) or maintain a sliding window $e - w + 1$ to e for some window size w.

5.3.2 Indistinguishability

Because the proof is embedded in an ordinary MLS `PrivateMessage`-encrypted and authenticated—the hub cannot distinguish it from a normal application message. Therefore, it cannot selectively drop proofs messages.

5.3.3 Detection Probability

The detection probability using probabilistic proof sampling is analyzed as follows:

Proof-request Probability per Message Given n clients, each independently requesting a proof with probability α:

$$P_{\text{proof}} = 1 - (1 - \alpha)^n.$$

Detection Probability in a Single Message The hub attacks each message with probability β. Hence:

$$P_{\text{detect}}^{(1)} = \beta[1 - (1 - \alpha)^n].$$

Escape Probability over T Messages Assuming independence across messages:

$$P_{\text{escape}}^{(T)} = \left[1 - \beta(1 - (1 - \alpha)^n)\right]^T.$$

Detection Probability by Message T

$$\boxed{P_{\text{detect}}^{(T)} = 1 - \left[1 - \beta(1 - (1 - \alpha)^n)\right]^T.}$$

5.3.4 Sampling Rate Selection

We show in Chapter 6 that choosing $\alpha \approx 0.05$ yields a 95 % detection probability within $T = 40$ messages when group size $n = 100$ and hub-attack probability $\beta = 0.10$. Smaller or larger groups and different risk tolerances can adjust α accordingly.

5.4 Client-Side Verification Process

When any client C' receives an `AuditProof`-carrying MLS message from another client C, it executes the following verification routine:

Algorithm 2 OnReceiveProof(R, $t_{\max}$, s, e)

Decrypt the MLS `PrivateMessage` to extract $(R, t_{\max}, s, e)$.
Reconstruct local leaves $\ell_i = H(d_i \,\|\, t_i)$ for $i = s, \ldots, e$.
$\hat{R} \leftarrow$ RecomputeMerkleRoot($\{\ell_s, \ldots, \ell_e\}$.
$t'_{\max} \leftarrow \max\{t_i : i = s, \ldots, e\}$.
if $\hat{R} \neq R$ or $t'_{\max} \neq t_{\max}$ **then**
    RaiseAlert(ProofMismatch, C).
    Broadcast an integrity-violation message to the room.
end if
Broadcast $\hat{R}$ to the Room

If a mismatch is detected—i.e., $\hat{R} \neq R$ or the timestamps disagree—then at least one of the messages in $[s, e]$ was dropped, delayed, or reordered by the hub. The client then immediately issues its own proof broadcast (to ensure consensus among honest peers) and raises a local alert.

5.5 Conditions for Detecting a Malicious Hub

Clients monitor message sequences and hub behavior continuously. A hub is considered malicious if any of the following conditions occur:

1. **Proof Mismatch:** A received Merkle proof does not match the locally computed proof, indicating message reordering, dropping, or tampering by the hub.

2. **Timestamp Skew:** Message timestamps provided by the hub either regress ($t_{i+1} < t_i$) or significantly jump forward by more than a permissible bound.

3. **Conflicting Delivery Report:** Two delivery reports issued by the hub are not monotonically increasing in timestamp order, suggesting possible reordering or selective message suppression.

4. **Local Client History Inconsistency:** A client observes that its own consecutively sent messages have been reordered by the hub. Specifically, if a client sends messages m_1 followed by m_2, but receives a delivery report or observes a message sequence where m_2 appears delivered before m_1, it marks the hub as malicious.

When any of these conditions trigger, the client immediately alerts other group participants to the malicious behavior by broadcasting the proof mismatch or inconsistency evidence, enabling group-wide detection and prompt response.

5.6 Implementation Overview

We built a proof-of-concept in Rust atop the `openmls` crate:

- **Message Store.** Each client maintains a local, ordered list of message tuples (d_i, t_i), where d_i is the message plaintext and t_i is the hub-assigned timestamp. Messages are appended as they arrive, and sorted by timestamp when generating the Merkle root.

- **Merkle Tree Module.** A Merkle tree is constructed over a range of messages indexed from s to e. We use SHA-256 as the cryptographic hash function. The resulting Merkle proof tuple, which is broadcasted, is structured as $(R,\ t_{\max},\ s,\ e)$.

- **MLS Integration.** Merkle proofs are encapsulated within encrypted `MLS PrivateMessage` frames and broadcasted to the group. Proof messages begin with a special prefix

`<MERKLE_PROOF>` to signal the client the content clearly. Importantly, this integration preserves the standard MLS handshake and delivery mechanisms without modification.

Example Overhead:

- **Regular MLS message** ("hello, alice"):
  Payload: 153 bytes, Plaintext size: 12 bytes
- **MLS message with Merkle proof**:
  Payload: 212 bytes, Plaintext size: 70 bytes

This completes the protocol design description. In Chapter 6 we quantify detection rates and computation overhead.

Chapter 6

Results

This chapter answers two questions:

1. *How fast do honest clients detect message-order tampering?*
2. *What memory and CPU overhead does our audit layer add?*

We report on 1 000-trial Monte-Carlo simulations and micro-benchmarks of a 100-node Rust prototype integrated with `OpenMLS`.

6.1 Detection Probability

Every simulation picks a first attack position at random, then lets the hub reorder subsequent messages with probability β. A run is *detected* once any honest client raises a ProofMismatch. Detection probability after T messages is therefore the fraction of runs caught no later than message T.

Figures 6.1, 6.2, 6.3, and 6.4 summarise various scenarious for computing detection probability.

Figure 6.1 plots cumulative detection vs. message index. The red dashed line marks the *average* first-attack position (9.5). Vertical dotted lines show that 90 % of attacks are caught

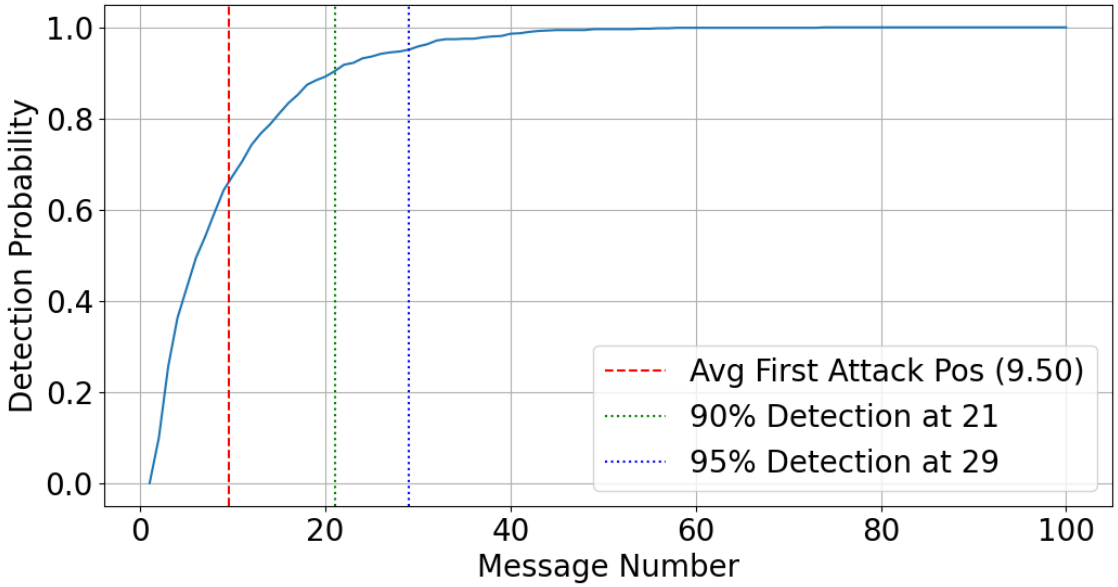

Figure 6.1: Detection probability for a single reordering attack ($\alpha = 0.05$, $\beta = 0.10$, $n = 100$).

by message 21 and 95 % by message 29. Even a modest sampling rate therefore yields rapid detection.

Figure 6.2 sweeps the hub's attack probability β. A more aggressive hub is discovered sooner merely because it tampers more often—not because the protocol itself changes.

Figure 6.3 shows diminishing returns beyond $\alpha = 0.05$.

Figure 6.4 highlights the aggregate-sampling effect. A 100-member room hits 95 % detection in ≈25 messages; a two-person chat needs ≈45.

Observations

- With $\alpha = 0.05$, $\beta = 0.10$, and $n = 100$, 95 % of attacks are detected within 30 messages (Fig. 6.1).
- Higher β accelerates detection (Fig. 6.2); lower values delay detection because attacks are rarer.

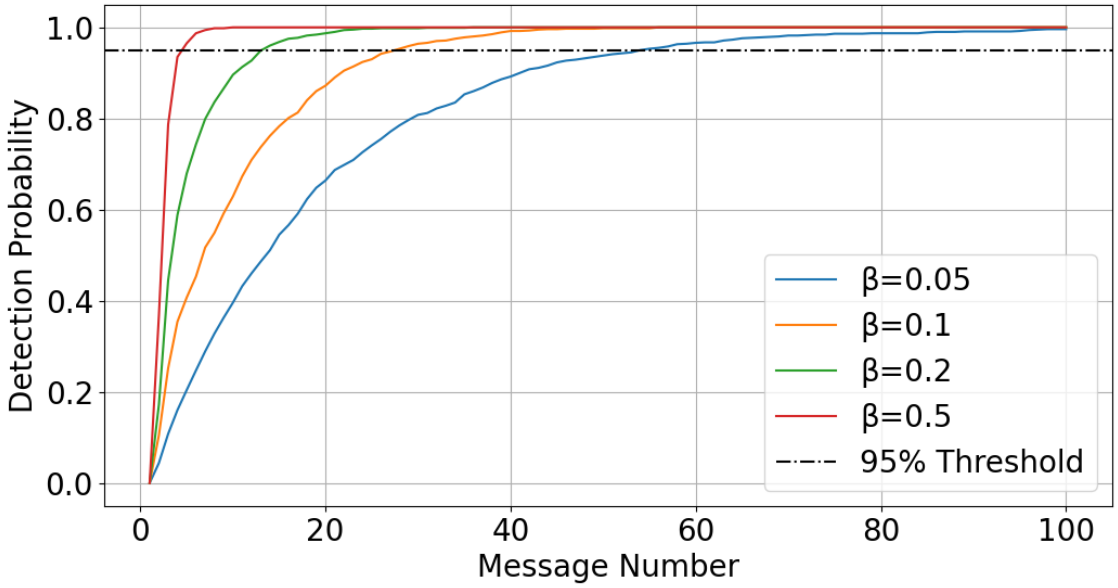

Figure 6.2: Impact of hub aggressiveness β ($\alpha = 0.05$, $n = 100$).

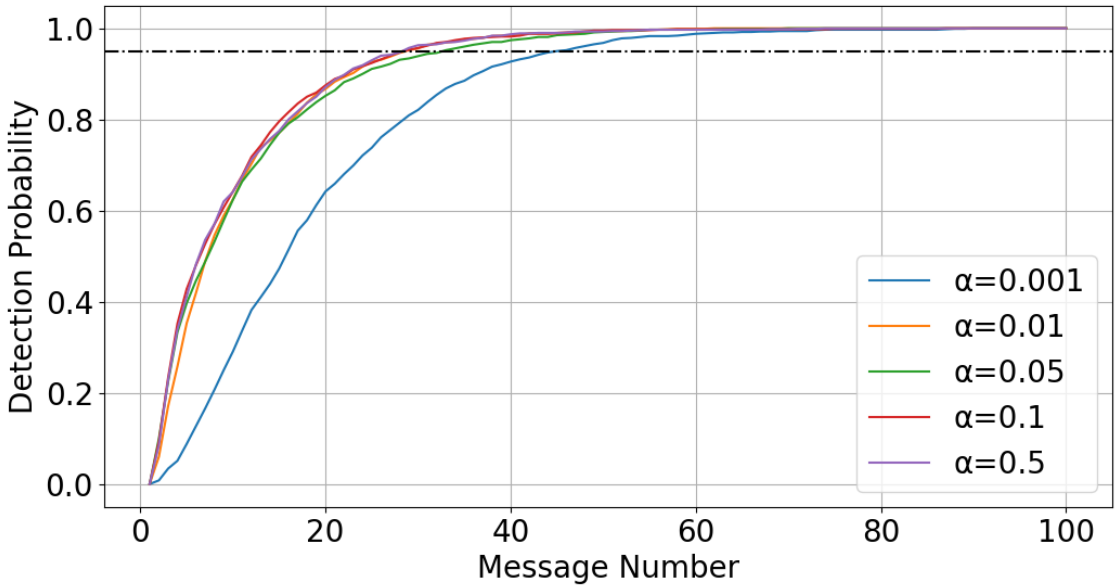

Figure 6.3: Effect of client sampling rate α ($\beta = 0.10$, $n = 100$).

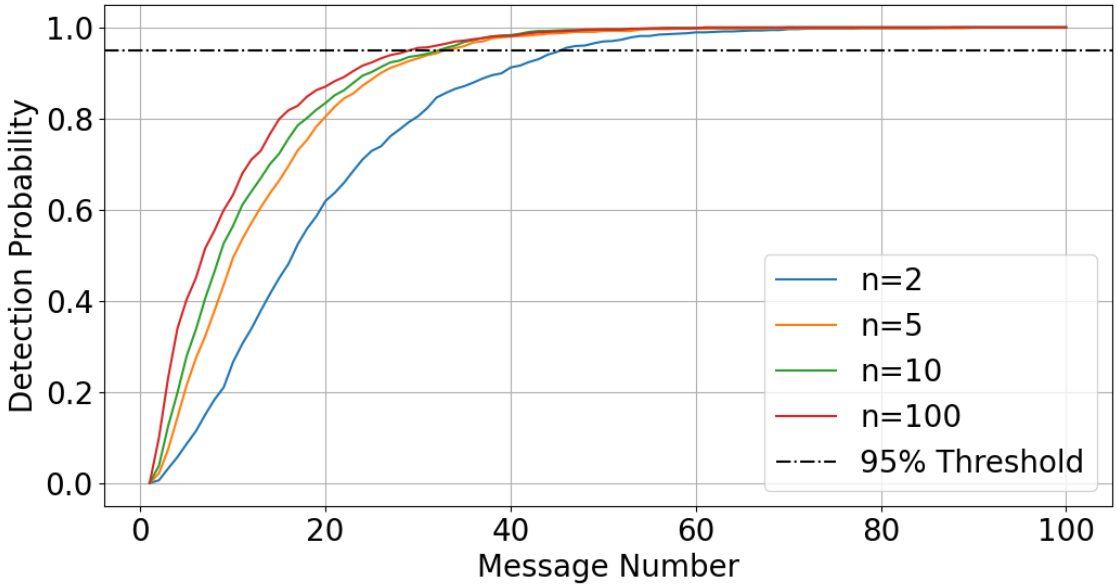

Figure 6.4: Detection probability vs. room size ($\alpha = 0.05$, $\beta = 0.10$).

- Increasing α from 0.001 to 0.01 dramatically shortens latency, but gains flatten above 0.05 (Fig. 6.3).

- Larger rooms detect faster thanks to the aggregate probability $1 - (1 - \alpha)^n$ (Fig. 6.4).

6.2 Performance Overhead

We evaluated both the computation and the memory overhead as a function of the size of the group. Additional memory is required at each client to maintain the local message order, which involves storing a sorted map of message content along with their corresponding hub-assigned timestamps. The computational overhead arises from generating the Merkle root and embedding the resulting inclusion proof within an encrypted `MLS PrivateMessage`. These experiments were performed using our open-source MIMI implementation written in Rust, which integrates the `OpenMLS` library to provide the MLS layer for end-to-end encrypted group communication and key management. The experiments were performed by emulating

our Rust client on a 2023 MacBook Pro (M1 Pro, 16 GB).

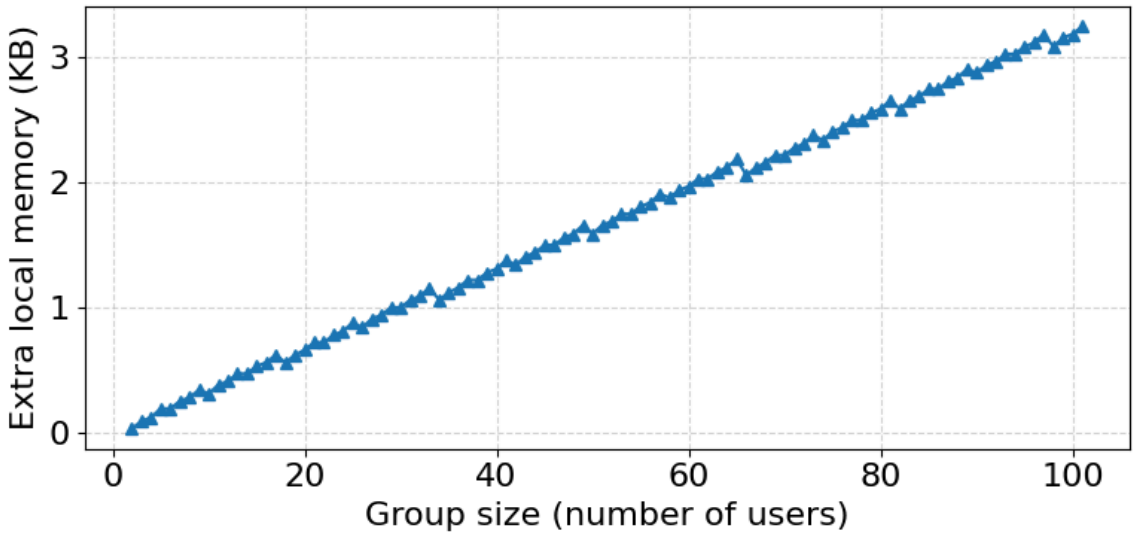

Figure 6.5: Extra client memory per group size.

Memory overhead As shown in Figure 6.5, the local memory required to store messages for computing Merkle proofs increases roughly linearly with group size, reaching just above 3 kB for $n = 100$.

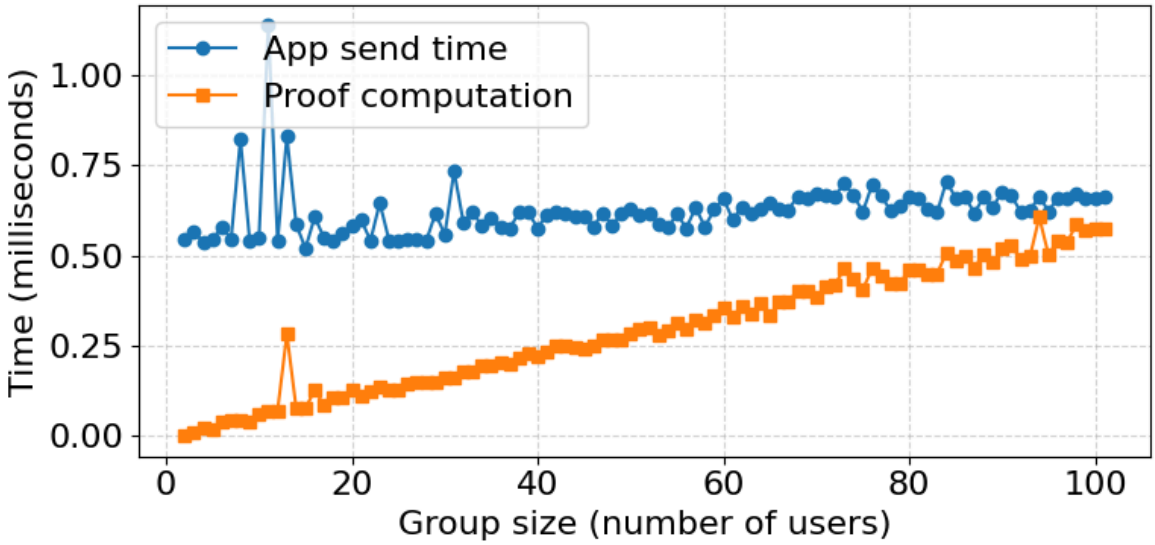

Figure 6.6: Median MLS send latency and proof-generation time.

Computation time Figure 6.6 plots the median time required by the client to send an encrypted `MLS PrivateMessage` as a function of group size, alongside the additional time required to compute a Merkle inclusion proof when a message is sampled. The application send time represents the time taken by the client to construct and send an encrypted MLS message. The proof computation time corresponds to the time required to locally compute the Merkle proof over the client's timestamp-ordered message list. We observe that the MLS message latency remains mostly constant around 0.6 ms, with occasional spikes due to initial setup. In contrast, Merkle proof computation grows approximately linearly with group size, adding at most 0.6 ms for $n = 100$.

6.3 Scalability Analysis

- **Detection vs. n.** Larger rooms benefit from aggregate sampling (Fig. 6.4); 95 % detection drops from 45 messages at $n = 2$ to 25 at $n = 100$.
- **Resource growth.** Memory and CPU overhead scale linearly with room size but remain well under 3 kB and under 1 ms for $n \leq 100$.

6.4 Discussion of Findings

- **Fast detection.** Even at $\alpha = 0.05$, hubs are exposed within 30 messages in 100-member rooms.
- **Sweet-spot sampling.** $\alpha = 0.01$–0.05 balances speed and overhead; raising α further yields diminishing benefit.
- **Lightweight cost.** The audit layer adds 3 kB memory and ≤ 1 ms latency per proof.

- **Recommended settings.**
  - *Small rooms* ($n \leq 5$): $\alpha = 0.10$
  - *Larger rooms* ($n > 5$): $\alpha = 0.05$
  - *Critical or regulated rooms*: $\alpha \geq 0.10$

Probabilistic Merkle auditing therefore delivers near-real-time integrity with minimal overhead, hence it is Dedeployable on mobile devices without hub changes.

Chapter 7

Conclusions & Future Work

7.1 Key Findings and Contributions

This thesis tackled a blind spot in interoperable, end-to-end–encrypted messaging: verifiable message-order integrity in the presence of an untrusted hub. The main contributions can be summarised as follows:

1. **Threat model.** We clarified the attack surface introduced by the MIMI hub's timestamp authority and showed that a single malicious provider can manipulate room transcripts without violating MLS confidentiality.

2. **Timestamp-Merkle audit layer.** A lightweight, client-side protocol detects any drop, delay, or re-ordering attack with $> 95\%$ probability after at most 40 subsequent messages.

3. **Prototype & evaluation.** A Rust implementation built on `OpenMLS` with < 0.6 ms proof latency and ≈ 3 kB extra RAM per client, confirming the practicality of the design.

7.2 Limitations

- **Static sampling rate.** The current prototype fixes the proof-broadcast probability α at compile-time. Adaptive adversaries could, in principle, learn this rate and tailor their behaviour.

- **No on-chain transparency.** Our audit proofs remain local to the room. They do not create a globally searchable log such as Certificate Transparency's Merkle forest [32].

- **Hub removal is manual.** Once misbehaviour is detected there is no existing mechanism to migrate to a new room.

7.3 Future Research Directions

7.3.1 Adaptive Malicious-Hub Defence

The prototype fixes the proof-broadcast probability at compile time, yet a determined adversary could eventually infer that rate and schedule attacks so that they slip through the detection window. A promising line of work is to let each client adjust dynamically in response to local observations such as sudden latency spikes or out-of-order message deliveries.

One approach is to model network or adversarial behavior as a time series and train a lightweight machine learning predictor (e.g., an LSTM or ARIMA model) on recent delivery metrics—latency, missing proof acknowledgments to forecast the likelihood of message tampering or reordering. Clients could then increase when the model signals higher risk (e.g., sudden timestamp deviations) and decrease it during stable periods to conserve bandwidth.

7.3.2 Automatic Group Reconciliation

Detection by itself does not restore trustworthy service; once a hub has been flagged, the room must migrate to a new provider. MLS already supports exter- nal commits that can remove members, rotate credentials, or even re-initialise the group.Building on this capability, one could design an automatic "hub-handover" transaction that expels the compromised hub and nominates a replacement without manual intervention.

7.3.3 Public-Safety Networks

Emergency-response infrastructure is still dominated by SIP and the Message Session Relay Protocol. The Content-External Media Attachment (CEMA) extension enables end-to-end object security for MSRP [19], suggesting a practical bridge between legacy PSAP systems and the HTTPS-based MIMI ecosystem. A gateway acting as a client in a MIMI group could translate messages to and from Public Safety Answering Points (PSAPs), enabling seamless communication between PSAP domains and regular MIMI users. Such a gateway would allow first-responder applications to benefit from MLS security without a replacing the existing NG-911 infrastructure.

7.3.4 Privacy-Preserving User Discovery

MIMI currently leaves discovery to provider-defined mechanisms, yet users often maintain separate work and personal personas and may wish to reveal as little metadata as possible. Future designs could combine multi-persona public keys with attribute-based encryption so that a directory discloses only the subset of identities authorised by the requester's credentials. Private information-retrieval techniques can used to achive this [12].

7.4 Concluding Remarks

MIMI is still a moving target at the IETF, but the work presented here shows how order-integrity—a property often overlooked in mainstream E2EE deployments—can be achieved with negligible overhead and without new trusted infrastructure. The timestamp-Merkle approach is protocol-agnostic- it can be integrated onto any central server design. As interoperability mandates mature and large providers prepare DMA-compliant applications, incorporating such verifiable ordering mechanisms will be crucial for maintaining user trust across provider boundaries. In short, while MIMI is far from production-ready, this thesis demonstrates a viable path toward dependable, cross-platform messaging.

Bibliography

[1] Regulation (eu) 2022/1925 on contestable and fair markets in the digital sector (digital markets act). Official Journal of the European Union, L 265/1, September 2022. URL https://eur-lex.europa.eu/eli/reg/2022/1925/oj. Accessed 29 May 2025.

[2] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer, 2019.

[3] David Balbás, Daniel Collins, and Phillip Gajland. Whatsupp with sender keys? analysis, improvements and security proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 307–341. Springer, 2023.

[4] Richard Barnes. An Architecture for More Instant Messaging Interoperability (MIMI). Internet-Draft draft-ietf-mimi-arch-01, Internet Engineering Task Force, November 2024. URL https://datatracker.ietf.org/doc/draft-ietf-mimi-arch/01/. Work in Progress.

[5] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022. URL https://www.rfc-editor.org/info/rfc9180.

[6] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023. URL https://www.rfc-editor.org/info/rfc9420.

[7] Richard Barnes, Matthew Hodgson, Konrad Kohbrok, Rohan Mahy, Travis Ralston, and Raphael Robert. More Instant Messaging Interoperability (MIMI) using HTTPS and MLS. Internet-Draft draft-ietf-mimi-protocol-03, Internet Engineering Task Force, March 2025. URL https://datatracker.ietf.org/doc/draft-ietf-mimi-protocol/03/. Work in Progress.

[8] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large dynamic groups. Internet-draft draft-barnes-mls-treekem-00, Internet Engineering Task Force, May 2018. URL https://hal.inria.fr/hal-02425247. Work in progress.

[9] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal models and verified protocols for group messaging: Attacks and proofs for ietf mls. Technical Report RR-9398, Inria Paris, Dec 2019. URL https://hal.inria.fr/hal-02425229.

[10] Vitalik Buterin. Verkle trees: Next-generation state for ethereum, 2022. URL https://vitalik.ca/general/2021/06/18/verkle.html.

[11] Ruiqi Cheng and et al. Talek: Private group messaging with hidden access patterns. In *Annual Computer Security Applications Conference (ACSAC)*, pages 84–99, Austin, USA, Dec 2020. doi: 10.1145/3427228.3427231.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[13] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1802–1819, 2018.

[14] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security*, 2009.

[15] Johannes Derler, Claudia Jentner, and Daniel Slamanig. Puncturable signatures and their applications in randomised integrity audits. In *ACM CCS*, 2018.

[16] A. Fábrega, C. O. Pérez, A. Namavari, B. Nassi, R. Agarwal, and T. Ristenpart. Injection attacks against end-to-end encrypted applications. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2648–2665, 2024. doi: 10.1109/SP54263.2024.00082.

[17] Git Development Community. Git – fast, scalable, distributed revision control system. URL https://git-scm.com. Accessed: [your access date].

[18] Robert L Harrison. Introduction to monte carlo simulation. In *AIP conference proceedings*, volume 1204, page 17, 2010.

[19] Christer Holmberg, Staffan Blau, and Eric Burger. Connection Establishment for Media Anchoring (CEMA) for the Message Session Relay Protocol (MSRP). RFC 6714, August 2012. URL https://www.rfc-editor.org/info/rfc6714.

[20] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303. IEEE, 2021.

[21] Shimin Jing, Xin Zheng, and Zhengwen Chen. Review and investigation of merkle tree's technical principles and related application fields. In *2021 International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, pages 86–90. IEEE, 2021.

[22] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. URL https://www.rfc-editor.org/info/rfc8032.

[23] Martin Kleppmann, Stephan A Kollmann, Diana A Vasile, and Alastair R Beresford. From secure messaging to secure collaboration. In *Security Protocols XXVI: 26th International Workshop, Cambridge, UK, March 19–21, 2018, Revised Selected Papers 26*, pages 179–185. Springer, 2018.

[24] Mallory Knodel, Sebastian Celi, Olaf Kolkman, and Georgia Grover. Definition of end-to-end encryption. IACR ePrint 2024/2085, 2024. URL https://eprint.iacr.org/2024/2085.

[25] Amos E. Komo and Marcos A. Simplicio Jr. Auditable messages with hash chains in instant messaging apps. In *XXIV Brazilian Symposium on Information and Computer System Security (SBSeg) – Extended Abstracts*, pages 25–32, 2024. doi: 10.5753/sbseg_estendido.2024.241723.

[26] Arun Krishnan. Secure and scalable group messaging with mls protocol. Bachelor's Thesis, Metropolia University, 2022. URL https://www.theseus.fi/handle/10024/748768.

[27] Chainguard Labs. Securing software supply chains with merkle-tree sboms, 2023. URL https://www.chainguard.dev/unchained/merkle-sbom.

[28] Dana Landis, Eugenia Bietti, and Sarah Park. Sok: "interoperability vs security" arguments: A technical framework. arXiv:2502.04538, 2025. URL https://arxiv.org/abs/2502.04538.

[29] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016. URL https://www.rfc-editor.org/info/rfc7748.

[30] Ben Laurie. Improving authenticated dictionaries. *IACR ePrint 2013/006*, 2013.

[31] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, 2013. URL https://www.rfc-editor.org/rfc/rfc6962.

[32] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, Internet Engineering Task Force, Jun 2013. URL https://www.rfc-editor.org/rfc/rfc6962.

[33] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. Understanding bittorrent: An experimental perspective. 2005.

[34] Mitsuru Matsui, Shiho Moriai, and Junko Nakajima. A Description of the Camellia Encryption Algorithm. RFC 3713, April 2004. URL https://www.rfc-editor.org/info/rfc3713.

[35] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008. URL https://www.rfc-editor.org/info/rfc5116.

[36] Ralph C. Merkle. *Secrecy, Authentication and Public Key Systems*. PhD thesis, Stanford University, 1979.

[37] National Institute of Standards and Technology. Advanced Encryption Standard (AES). FIPS PUB 197, November 2001. URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[38] National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-4, July 2013. URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[39] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018. URL https://www.rfc-editor.org/info/rfc8439.

[40] Erik Nygren and Ryan Sleevi. Certificate Transparency Version 2.0. RFC 9162, 2022. URL https://www.rfc-editor.org/rfc/rfc9162.

[41] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *GitHub wiki*, 112 (4), 2016.

[42] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. URL https://www.rfc-editor.org/info/rfc8446.

[43] Hector Sanjuan, Sanna Poyhtari, Pedro Teixeira, and Ignacio Psaras. Merkle-crdts: Merkle-dags meet crdts. arXiv:2004.00107, 2020. URL https://arxiv.org/abs/2004.00107.

[44] Martin Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability (mobile cowpi). IACR ePrint 2018/1097, 2018. URL https://eprint.iacr.org/2018/1097.

[45] Michael Schliep, Ian Kariniemi, and Nicholas Hopper. Is bob sending mixed signals? In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, WPES '17, page 31–40, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351751. doi: 10.1145/3139550.3139568. URL https://doi.org/10.1145/3139550.3139568.

[46] Adam Sutton and Reza Samavi. Timestamp-based integrity proofs for linked data. In *Proceedings of the International Workshop on Semantic Big Data*, pages 1–6, Houston, USA, 2018. doi: 10.1145/3208352.3208353.

[47] Kevin Thomas and et al. Robust, privacy-preserving, transparent, and auditable on-device blocklisting. arXiv:2304.02810, 2023. URL https://arxiv.org/abs/2304.02810.

[48] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the cryptographic deniability of the signal protocol. In *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II 18*, pages 188–209. Springer, 2020.

[49] Théophile Wallez, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: Authenticated group management for messaging layer security. In *USENIX Security Symposium*, 2023. URL https://www.usenix.org/conference/usenixsecurity23/presentation/wallez.