



ARTICLE

Ember: A Serverless Peer-to-Peer End-to-End Encrypted Messaging System with Per-Message Forward Secrecy over an IPv6 Mesh Network

Hamish Alsop^{1,*}, Leandros Maglaras^{1,2,*} and Naghmeh Moradpoor¹

¹School of Computing, Edinburgh Napier University, Edinburgh, UK

²School of Computing, De Montfort University, Leicester, UK

*Corresponding Authors: Hamish Alsop. Email: 40342915@napier.live.ac.uk; Leandros Maglaras.

Email: leandros.maglaras2@dmu.ac.uk

Received: 24 March 2026; Accepted: 17 August 2026; Published: 22 October 2026

ABSTRACT: A substantial body of research has focused on formalising what constitutes a “secure” messaging system, recognising that end-to-end encryption alone is insufficient to capture the full range of security, privacy, and usability properties that are expected by modern users. Several solutions have been proposed recently, each with their own drawbacks, making the need for a direct, infrastructure-light secure messaging system a continuing concern. This paper presents *Ember*, a serverless peer-to-peer messaging system providing end-to-end encrypted communication over a decentralised IPv6 mesh network. Ember combines an Extended Triple Diffie–Hellman (X3DH) initial key agreement with the Double Ratchet algorithm to provide per-message forward secrecy (FS) and post-compromise security (PCS) without any central key distribution service. Since per-message keys can remain in the MessageKeyStore after their associated ciphertext records are deleted, the forward secrecy and secure deletion guarantees are strictly bounded against a forensic adversary. The system operates without central servers, enforces data minimisation through ciphertext-only local storage and time-based message expiration, and prioritises architectural clarity, explicit trust boundaries, and practical deployability on Android. The paper describes the system architecture, cryptographic design, network model, and security properties. Ember includes a ciphertext-only persistence model using an encrypted local database, ensuring that message plaintext is never written to disk, and integrates a time-to-live (TTL) mechanism that reliably deletes expired ciphertext records within approximately 100 ms of their nominal expiry. Ember is a layered and analysable system architecture with explicit trust boundaries separating user interface logic, cryptographic operations, storage, and networking components. The paper presents a controlled dynamic evaluation that correlates instrumented application telemetry with link-layer packet capture, demonstrating that the Double Ratchet is correctly driven by direction-reversal-triggered Diffie–Hellman ratchet steps, that the verify-then-decrypt processing order is upheld for every inbound message, and that the Yggdrasil link layer presents an opaque, fixed-size, fixed-cadence traffic profile to a relay-adjacent observer. The evaluation also surfaces a precise residual forward-secrecy gap in the current implementation which is documented as a targeted remediation item. The paper closes by discussing limitations and future work.

KEYWORDS: End-to-end encryption; secure messaging; peer-to-peer systems; forward secrecy; double ratchet; X3DH; decentralised networking; mobile security; data minimisation; IPv6 mesh

1 Introduction

End-to-end encrypted (E2EE) messaging has become foundational to modern digital communication, but its widespread deployment has attracted increasing regulatory attention. Proposed legislative initiatives,

most notably the European Union’s “Chat Control” framework, seek to introduce mechanisms for lawful access to private communications, raising serious concerns within the security research community about the long-term viability of strong E2EE guarantees [1].

Such proposals typically rely on client-side content scanning or mandatory interception capabilities that, if deployed, would negate the end-to-end security model by introducing surveillance at the endpoint. From a security engineering perspective, these approaches reintroduce systemic vulnerabilities, create new attack surfaces, and erode the trust assumptions on which secure messaging is built [2].

In parallel, most widely deployed messaging platforms rely on centralised architectures where message routing, metadata storage, and key management are mediated by service providers. Even where encryption protects message contents, users must place substantial trust in these providers and in the legal environments in which they operate. Centralisation introduces single points of failure, including both technical and institutional, and concentrates the risk of large-scale surveillance, data retention, or compelled access.

This work is motivated by two questions. First, can a secure messaging system be constructed in a manner that meaningfully resists regulatory and infrastructural pressures on end-to-end encryption while still providing the strong compromise-resilience guarantees expected of a modern messenger? Second, can reliance on centralised service providers be reduced or eliminated entirely through decentralised, serverless communication? By removing central servers and minimising persistent data on end-user devices, this project explores an alternative design space that prioritises user autonomy, data minimisation, and architectural resilience, without sacrificing per-message forward secrecy and post-compromise security.

The objective is not to deploy a mass-market platform, but to rigorously examine whether decentralised and serverless secure messaging, equipped with ratcheting-based key evolution, remains technically viable under tightening legislative and infrastructural constraints.

1.1 Problem Statement

The problem addressed in this paper is the design and evaluation of a secure messaging system that operates without centralised infrastructure, minimises retained user data, and maintains end-to-end confidentiality with continuous key evolution on mobile platforms. In the context of emerging regulatory pressures on encrypted communication, there is a need to investigate architectures that avoid the traditional client-server model entirely and thereby reduce the applicability of legislation that assumes the presence of service providers or central intermediaries.

Specifically, this work examines whether peer-to-peer secure messaging can be achieved in a practical and analysable manner using direct device-to-device communication over a decentralised IPv6 mesh network, while integrating a ratcheting key-evolution protocol that ordinarily depends on server-mediated pre-key distribution. The absence of central servers eliminates conventional message queues, metadata aggregation points, provider-controlled storage, and, critically, the pre-key servers on which asynchronous key agreement normally relies, fundamentally altering the threat model and trust assumptions compared to mainstream messaging applications.

Additionally, mobile platforms impose unique constraints, including background execution limits, restricted networking capabilities, and persistent storage risks. Any viable solution must therefore operate within these constraints while still enforcing strong security properties, such as authenticated encryption, per-message key evolution, encrypted storage, ephemeral message retention, and explicit identity management.

The Ember system is designed to explore this problem space by implementing a serverless, peer-to-peer messaging architecture that provides end-to-end encryption with per-message forward secrecy

and post-compromise security, encrypted local persistence, and automatic message expiration. The system intentionally limits supported features, such as multimedia transfer and group messaging, in order to prioritise security clarity, correctness, and minimisation of retained data.

1.2 Research Objectives

The primary objective of this research is to assess whether the implemented Ember system satisfies its intended security and architectural goals under realistic operating conditions. Rather than proposing a novel cryptographic primitive, this work focuses on system-level design, integration, and validation of established cryptographic mechanisms, specifically the X3DH handshake and the Double Ratchet algorithm, within a decentralised mobile environment that lacks the coordinating infrastructure these mechanisms were originally designed around.

The research objectives are as follows. First, to evaluate the confidentiality and integrity of message data transmitted over a peer-to-peer IPv6 mesh network, ensuring that message contents remain protected against network-level adversaries. Second, to validate that message data is securely stored at rest using encryption and that plaintext is not persistently retained on the device beyond volatile memory usage. Third, to assess the effectiveness of the system's time-to-live (TTL) mechanism in enforcing automatic message deletion and supporting data minimisation goals. Fourth, to verify, against runtime evidence, that the Double Ratchet is correctly integrated into the live message-processing pipeline, that direction-reversal-triggered Diffie-Hellman ratchet steps occur reliably, that the previous-chain-length accounting is propagated correctly, and that per-message key evolution actually takes place rather than silently degenerating into a static-key mode.

Further objectives include examining the practicality of identity verification via cryptographic key fingerprints in a serverless context, and evaluating the operational reliability of peer-to-peer communication over a decentralised IPv6 mesh network. In particular, this work seeks to determine whether such a networking model is a viable alternative to traditional server-mediated message delivery or whether it introduces prohibitive complexity or reliability challenges.

Importantly, while this research targets the implementation and runtime validation of ratcheting-based forward secrecy (FS) and post-compromise security (PCS), it does not claim a complete end-to-end empirical validation of the X3DH handshake within a single capture window, nor does it claim that the forward-secrecy property is delivered without qualification. As reported in the evaluation, the captured conversation pre-existed the test session, so the X3DH handshake itself is described from the implementation rather than exercised on the wire; and the current build retains per-message key material in a dedicated key store after the corresponding ciphertext has been deleted, qualifying the ephemerality of key state. These are treated as precisely bounded, remediable gaps rather than as fundamental design limitations.

1.3 Contributions

This paper makes the following concrete and implemented contributions:

- The design and implementation of a functioning serverless peer-to-peer messaging system that provides end-to-end encrypted communication over a decentralised IPv6 mesh network.
- The integration of an Extended Triple Diffie-Hellman (X3DH) initial key agreement with a Double Ratchet per-message key evolution protocol, providing forward secrecy and post-compromise security on a per-message basis without reliance on any central key distribution service, together with runtime evidence that the ratchet is correctly driven under live operating conditions.
- A ciphertext-only persistence model using an encrypted local database, ensuring that message plaintext is never written to disk.

- The integration of a time-to-live (TTL) mechanism for automatic message expiration that reliably deletes expired ciphertext records within approximately 100 ms of their nominal expiry, enforcing application-level data minimisation on mobile devices.
- A layered and analysable system architecture with explicit trust boundaries separating user interface logic, cryptographic operations, storage, and networking components, together with a controlled dynamic evaluation correlating instrumented application telemetry with link-layer packet capture.

These contributions collectively demonstrate the feasibility of decentralised, serverless secure messaging on mobile platforms with strong, per-message compromise-resilience guarantees, while maintaining clear and defensible security boundaries.

The remainder of this paper proceeds as follows. [Section 2](#) reviews related work. [Section 3](#) defines the threat model. [Sections 4–6](#) describe the system architecture, network design, cryptographic mechanisms, key lifecycle, data persistence, and message processing pipelines. [Section 6.4](#) analyses privacy and metadata considerations. [Section 7](#) presents security testing and verification results, including the dynamic ratchet-validation evaluation. [Section 8](#) evaluates the system and discusses design trade-offs. [Section 9](#) addresses limitations and future work, and [Section 10](#) concludes.

2 Related Work and Background

2.1 Secure Messaging Threat Models and Security Objectives

A substantial body of research has sought to formalise what actually constitutes a “secure” messaging system, recognising that end-to-end encryption alone is insufficient to capture the full range of security, privacy, and usability properties that are expected by modern users. One of the most influential contributions in this area is the systematization of knowledge (SoK) on secure messaging by Unger et al. [3], which establishes a structured taxonomy of security goals, threat models, and design trade-offs across contemporary messaging applications.

The SoK identifies a set of cryptographic and system-level objectives that are now widely treated as baseline requirements. These include confidentiality and integrity under an active network adversary, authentication of communicating parties, forward secrecy (FS) to limit retrospective compromise, post-compromise security (PCS) to enable recovery after key exposure, and varying notions of deniability to prevent cryptographic transcripts from serving as verifiable proof of authorship. The authors emphasise that no deployed system simultaneously achieves all desirable properties. Secure messaging design is therefore framed as an exercise in prioritisation, where stronger guarantees in one dimension often introduce complexity, usability costs, or new failure modes elsewhere.

Beyond payload confidentiality, the SoK explicitly elevates metadata to a first-class privacy concern. Even when message contents are end-to-end encrypted, information such as communication patterns, timing, peer relationships, and online presence could still remain observable to service providers or network adversaries. The work distinguishes between content security and metadata privacy, noting that most widely deployed systems strongly prioritise the former while offering limited protection for the latter. This distinction is particularly relevant for decentralised and peer-to-peer systems, where reducing central trust can introduce new forms of observability at the network or transport layers.

More recent work has reinforced and extended these observations. A 2025 systematic literature review of secure instant messaging applications [4] surveys a broad range of academic and applied systems, with particular emphasis on forensic artefacts, data persistence, and post-compromise recoverability. Although it is more motivated by forensic analysis, the review provides a comprehensive comparative overview of modern secure messaging architectures and highlights recurring gaps between theoretical security claims

and real-world behaviour. Notably, it shows that many systems advertising ephemerality or “disappearing messages” continue to leave recoverable artefacts at the application, operating system, or storage layers, complicating strong claims around data minimisation and deletion. As reported later in this paper, Ember’s own dynamic evaluation surfaces a closely related artifact which is the retention of per-message key state after ciphertext deletion. This artifact directly illustrates the gap between an ephemerality claim and its realization in code.

These works collectively establish a practical analytical framework for evaluating secure messaging systems: content confidentiality must be weighed alongside compromise resilience, metadata exposure, architectural trust assumptions, and platform-level constraints. This framework directly informs Ember’s design and evaluation. Ember pursues high-assurance content confidentiality with per-message forward secrecy and post-compromise security, reduced reliance on centralised infrastructure, and aggressive application-level data minimisation. It does not pursue comprehensive metadata anonymity. The choice of a serverless, peer-to-peer architecture reflects an explicit trade-off: decentralisation reduces dependence on trusted servers but introduces challenges related to routing visibility, availability, and peer discovery. Situating Ember within these established threat models allows its design decisions to be evaluated transparently against prior work rather than in isolation.

2.2 End-to-End Encryption Protocols and Ratcheting-Based Security

Modern secure messaging systems predominantly derive their cryptographic strength from the Signal protocol family, which combines an asynchronous authenticated key exchange with a continuous key-evolution mechanism known as the Double Ratchet. At a high level, the Signal design decomposes into two principal components: the Extended Triple Diffie–Hellman (X3DH) handshake, which establishes an initial shared secret between parties that may not be simultaneously online, and the Double Ratchet protocol, which incrementally evolves cryptographic state to provide forward secrecy (FS) and post-compromise security (PCS) during message exchange. Ember adopts precisely this two-stage structure including the X3DH for initial key agreement followed by a Double Ratchet for all subsequent message encryption. Moreover, the extensive formal literature surveyed below serves as the design and analysis basis for Ember’s cryptographic core rather than as a justification for deferring it.

Due to its central role in providing strong compromise resilience, the Double Ratchet has been the subject of extensive formal analysis. Collins et al. [5] provide a tight security analysis of the protocol in the multi-session setting, addressing limitations in earlier proofs where security bounds degraded with the number of epochs or concurrent sessions. By decomposing the Double Ratchet into modular primitives that includes continuous key agreement (CKA) and forward-secure authenticated encryption (FS-AEAD), they clarify which cryptographic assumptions underpin Signal’s recovery guarantees following state exposure. Importantly, the analysis also shows that achieving tight reductions under standard Diffie–Hellman assumptions remains challenging, particularly at the scale and concurrency levels seen in real-world deployments.

Complementary work has examined ratcheting protocols using logic-based formal methods. Li et al. [6] apply Logic of Events Theory (LoET) to model both the X3DH handshake and the Double Ratchet phases of Signal, demonstrating that security guarantees are tightly coupled across protocol stages. Xiao et al. [7] generalise this approach to reason about ratcheting protocols as a class, introducing dedicated abstractions for key evolution, causality, and authentication. Collectively, these works highlight that ratcheting protocols are difficult to verify compositionally: guarantees provided by later stages depend critically on assumptions made during initial key establishment, and cannot be inferred just by analysing individual components in isolation. This coupling directly motivates Ember’s evaluation strategy, which validates the live behaviour of the

composed protocol (X3DH-established root key plus Double Ratchet evolution plus envelope authentication) rather than only its constituent primitives.

The security of ratcheting-based designs is therefore inseparable from the properties of the underlying authenticated key exchange. X3DH remains the dominant design for asynchronous secure messaging, but has received comparatively less focused analysis. Hashimoto et al. [8] recast X3DH as a Signal-conforming authenticated key exchange and present a generic construction based on standard primitives, including post-quantum key encapsulation mechanisms. Their analysis exposes fundamental trade-offs between efficiency, forward secrecy, and deniability, showing that design choices such as reuse of handshake material, which are made for better performance and easier deployability, can measurably weaken forward secrecy. Ember's X3DH implementation uses long-term identity, signed pre-key, and (where available) one-time pre-key contributions over Curve25519, consistent with this analysis.

Deniability introduces a further axis of complexity. Unger and Goldberg [9] show that achieving strong offline and online deniability in authenticated key exchanges often conflicts with resistance to key-compromise impersonation attacks and practical deployability constraints. More broadly, this body of work demonstrates that forward secrecy, post-compromise security, deniability, efficiency, and implementation complexity cannot all be maximised simultaneously; real-world systems inevitably need to make explicit trade-offs between these properties.

The practical upshot is clear: while ratcheting-based designs provide exceptionally strong guarantees under well-defined threat models, they rely on intricate state machines, strict key-management assumptions, and carefully coordinated interactions between protocol stages. Incorporating a Double Ratchet into a new system, particularly one with a decentralised or serverless architecture that lacks pre-key servers and reliable delivery infrastructure, is non-trivial and demands rigorous re-evaluation of threat models, trust assumptions, and failure modes. Ember confronts these challenges directly: it adopts the X3DH-plus-Double-Ratchet design but must establish initial key material without a central pre-key server and must tolerate the message loss, reordering, and connection churn characteristic of its per-connection peer-to-peer transport. The formal literature on Signal-style ratcheting therefore functions in this paper both as the specification Ember implements against and as the analytical benchmark against which its residual gaps, and especially the per-message key-state retention surfaced in [Section 7](#), are measured.

2.3 Group Messaging and Multi-Device State

While two-party secure messaging protocols such as Signal can provide strong guarantees through continuous ratcheting, extending these guarantees to group messaging and multi-device scenarios introduces substantially greater complexity. Group conversations are inherently dynamic: participants may join or leave at arbitrary times, users may operate multiple devices concurrently, and cryptographic state must remain consistent across all active members. Achieving forward secrecy (FS) and post-compromise security (PCS) under these conditions requires scalable mechanisms for group membership management, key evolution, and authentication.

The Messaging Layer Security (MLS) protocol represents the current consensus solution to these challenges. Standardised by the IETF as RFC 9420 [10], MLS specifies an asynchronous group key establishment protocol designed to support groups ranging from a small number of participants to thousands, while maintaining FS and PCS across membership changes. Its design replaces ad hoc sender-key distribution with a structured tree-based group state, allowing cryptographic material to be updated efficiently as group membership evolves.

Importantly, MLS is not a self-contained messaging system. The MLS Architecture [10] makes explicit that secure deployment depends on additional system components, including an authentication service for binding identities to credentials and a delivery service for reliable message dissemination. Correct operation further relies on application-level logic to handle group consistency, access control, and recovery from partial failures. Subsequent work, such as the formally verified TreeSync protocol by Wallez et al. [11], demonstrates that many historical weaknesses in group messaging systems arise not from cryptographic primitives themselves, but from underspecified group state management semantics. While this work strengthens confidence in MLS as a standard, it also illustrates the significant implementation and verification complexity involved.

These assumptions complicate the integration of MLS into fully decentralised or serverless environments. MLS presumes globally consistent group state, dependable message delivery, and a mechanism for authenticating long-term identities, properties that are straightforward to realize in centralized or federated systems, but considerably more difficult to achieve without reintroducing coordinating infrastructure. In peer-to-peer settings, challenges such as state synchronisation, denial-of-service resistance, and identity binding become first-order concerns rather than implementation details. These are precisely the challenges Ember already confronts in the two-party case for its Double Ratchet, and they would be amplified in a group setting.

MLS accordingly serves as a benchmark for any future group extension rather than an immediate implementation target. Ember currently focuses on one-to-one communication over a serverless peer-to-peer overlay, prioritising minimal trusted infrastructure, reduced retained state, and analysable security properties. Supporting MLS-style group messaging or multi-device consistency would require substantial re-evaluation of Ember's trust assumptions and architectural goals. As such, group messaging is treated as explicit future work, with MLS providing a well-defined benchmark against which any future extensions can be evaluated.

2.4 Serverless and Peer-to-Peer Secure Messaging Systems

A smaller but influential body of work explores secure messaging systems that avoid centralised servers entirely, instead relying on direct peer-to-peer communication or minimal supporting infrastructure. These systems are typically motivated by strong censorship-resistance and metadata minimisation goals, and therefore provide the closest architectural comparison points to Ember.

Ricochet is one of the earliest practical implementations of serverless secure messaging. By leveraging Tor onion services as stable, pseudonymous endpoints, Ricochet enables direct peer-to-peer communication without revealing network identifiers or relying on central message relays. Kirsh's analysis of Ricochet's message-layer security [12] highlights both the strengths and limitations of this approach. While Tor provides transport-level anonymity and protects against local network observers, early Ricochet designs relied primarily on channel-layer encryption, leaving message content dependent on the security properties of the onion service protocol itself. Efforts to introduce application-layer encryption expose a fundamental tension: incorporating modern asynchronous protocols such as Signal would require key directories or pre-key publication, reintroducing metadata leakage and trusted infrastructure that Ricochet explicitly seeks to avoid. This tension is directly relevant to Ember, which integrates X3DH pre-key publication and Double Ratchet evolution while attempting to avoid a persistent central pre-key directory; Ember's current approach is to publish and exchange pre-key bundles directly between peers rather than through a global directory.

Cwtch extends this design space by addressing Ricochet's lack of asynchronous messaging. Lewis [13] proposes the use of untrusted, discardable relay servers that broadcast encrypted messages to all connected participants, enabling offline delivery without revealing sender–receiver relationships. While this approach preserves strong metadata resistance, it does so at the cost of increased bandwidth usage, broadcast

semantics, and more complex availability and denial-of-service assumptions. Cwtch illustrates that limited infrastructure can coexist with strong privacy goals, but only by accepting significant performance and complexity trade-offs. Cwtch's discardable-relay model is a natural reference point for any future Ember extension that seeks asynchronous pre-key distribution or offline delivery without reintroducing a trusted server.

Mesh-based messengers such as Briar further generalise peer-to-peer communication by supporting multiple transports, including Bluetooth, Wi-Fi, and Tor, allowing messages to propagate opportunistically across disconnected networks. Blöchliger and von Seck [14] classify such systems as client mesh networks, where end-user devices both originate and route messages. Briar exemplifies this model by synchronising encrypted message state across contacts when connectivity becomes available, providing resilience against infrastructure failure and censorship. However, this flexibility introduces substantial state management challenges, particularly for group communication, consistency, and key evolution.

These challenges are underscored by Song's cryptographic analysis of Briar [15], which identifies weaknesses related to handshake forward secrecy, denial-of-service resilience, and group message duplication. This analysis illustrates a recurring pattern across serverless and mesh-based messengers: eliminating central servers often necessitates bespoke cryptographic constructions and custom protocol logic, increasing both implementation complexity and verification difficulty. Ember's experience reinforces this pattern; its serverless integration of an otherwise standard ratchet surfaces a non-obvious implementation gap (per-message key-state retention after ciphertext deletion) that does not arise in the same form in server-mediated deployments.

Across Ricochet, Cwtch, and Briar, common design tensions emerge. Systems that prioritise serverlessness and metadata resistance frequently sacrifice low-latency delivery, global availability, or seamless multi-device support. Discovery mechanisms are typically out-of-band, presence information is limited, and group messaging scales poorly without introducing additional infrastructure. Nonetheless, these systems demonstrate that meaningful secure communication without central servers is achievable, provided that trade-offs are made explicit and carefully bounded.

Ember draws directly from this lineage while occupying a distinct point in the design space. Rather than relying on Tor onion services or ad hoc mesh routing, Ember leverages an encrypted IPv6 overlay network to provide stable peer addressing and transport confidentiality. This approach retains the benefits of direct peer-to-peer communication while simplifying routing semantics and avoiding broadcast or store-and-forward relays. Combined with an Android-first implementation, full X3DH-plus-Double-Ratchet key management, and aggressive application-level data minimisation, Ember positions itself as a pragmatic evolution of prior serverless messaging systems rather than a wholesale replacement for federated or centralised platforms.

2.5 Decentralised IPv6 Overlay Networking and Yggdrasil as an Enabling Substrate

A defining characteristic of Ember is its reliance on an encrypted IPv6 overlay network rather than a traditional client-server transport stack or a Tor-like anonymity system. This places the system within a small but growing body of work that explores decentralised, key-derived addressing and self-organising routing as an alternative to centrally coordinated communication infrastructures.

Early exploratory work in this space includes an engineering evaluation of Matrix federation over Yggdrasil by Floury at EPFL [16]. Although not peer reviewed, the report provides a practical assessment of how a complex messaging protocol behaves when deployed over a cryptographically addressed overlay network. The study highlights several advantages of Yggdrasil, including automatic peering, encrypted links, stable key-derived IPv6 addressing, and improved connectivity in NAT-restricted or hostile network

environments. At the same time, it explicitly notes limitations such as routing convergence delays, sensitivity to topology changes, and the absence of built-in metadata-resistance properties. These observations reinforce the view that such overlays function primarily as transport substrates rather than as comprehensive privacy systems.

A more systematic evaluation of Yggdrasil's routing properties is provided by Pestov and Kyrychek [17]. Their large-scale experiments examine scalability, routing behaviour, and resource consumption across emulated topologies ranging from tens to hundreds of nodes. The results show that Yggdrasil maintains a modest and predictable memory footprint and supports long multi-hop paths without imposing a practical hop limit, making it viable for resource-constrained devices. However, the evaluation also identifies trade-offs inherent to its spanning-tree-based routing model. Performance characteristics depend on root node placement, and topology dynamics can introduce routing inconsistencies if root selection is unstable. Proposed mitigations, such as deliberately generating and pinning a stable root node, operate at the deployment level rather than being enforced by the protocol itself.

This body of work positions Yggdrasil as a capable but deliberately limited substrate for decentralised communication. It provides encrypted point-to-point transport, globally unique and collision-resistant addressing derived from cryptographic keys, automatic peer discovery, and reasonably stable routing under semi-static conditions. However, it does not attempt to provide metadata privacy, unlinkability, cover traffic, or resistance to traffic-analysis adversaries. Any such guarantees must therefore be implemented at higher layers, if required. As the dynamic evaluation in this paper shows empirically, Yggdrasil's link-layer behaviour does, as a side effect, present a fixed-size, fixed-cadence traffic profile to a relay-adjacent observer that incidentally obscures per-message activity; this is a useful but unintentional property and is not relied upon as a security guarantee.

For Ember, Yggdrasil functions as an enabler rather than as a security primitive. Its key-derived addressing model aligns cleanly with Ember's peer identity and binding mechanisms, and its decentralised routing eliminates the need for central servers, bootstrap infrastructure, or onion-service coordination. Ember deliberately does not treat the overlay as a source of anonymity or metadata protection: adversaries with visibility into overlay entry and exit points may still infer communication relationships and timing patterns. By explicitly framing Yggdrasil as a reachability and transport mechanism rather than a privacy enhancement, Ember adopts a conservative security posture that avoids conflating network connectivity with resistance to traffic analysis.

In summary, Yggdrasil occupies a distinct niche in the design landscape. It provides the connectivity foundation required for a serverless, mobile-first secure messenger without incurring the engineering overhead of Tor or the operational complexity of classical mesh routing protocols. Existing experimental and academic work validates its feasibility and scalability while clearly delineating its limitations. Ember builds upon this substrate to explore a middle ground between peer-to-peer security and practical deployability, acknowledging that stronger metadata-resistance mechanisms remain an area for future work.

2.6 Ephemerality, Deletion Semantics, and Data Remanence on Android

Ephemeral messaging features such as message timers, automatic purging of local state, and enforced deletion windows have become a prominent part of secure messaging user experience. However, extensive empirical evidence across Android versions demonstrates that logical deletion at the application layer does not imply physical secure deletion at the storage layer. For a system such as Ember, which aims to minimise retained plaintext and enforce short-lived visibility of messages, this literature imposes a fundamental constraint: TTL-based deletion can remove application-level access to content, but cannot guarantee eradication of underlying data traces under realistic forensic adversaries. Moreover, as Ember's

own evaluation demonstrates, application-level deletion must be applied symmetrically to all sensitive artefacts, both ciphertext records and the per-message key material that can decrypt them, or the ephemerality property is only partially realised.

Foundational work on Android data remanence by Vidas et al. [18] demonstrates that deleted application data frequently persists across multiple system locations, including filesystem journals, app-specific directories, and system service caches. Even after application uninstallation, artefacts may remain accessible through orphaned inodes, cached credentials, or residual inter-process state. The authors show that such residues can be exploited to recover sensitive material or support follow-on attacks, directly challenging assumptions that purging application storage meaningfully eliminates forensic artefacts.

Subsequent filesystem-level analysis reinforces this conclusion. Reardon et al. [19] show that Android's deletion primitives on `ext4` do not reliably clear inode metadata, directory entries, or journaling structures. Their experiments demonstrate that substantial portions of deleted data can remain recoverable even after prolonged device usage, and that file-carving techniques can reconstruct deleted SQLite databases and chat artefacts in the absence of filesystem metadata. More recent forensic analysis by Anglano [20] confirms that similar weaknesses persist on Android 9 and 10, with deleted artefacts frequently recoverable from journaling regions, write-ahead logs, and unallocated blocks despite the presence of TRIM and full-disk encryption mechanisms.

Beyond raw storage-level remanence, empirical studies of real-world messaging applications reveal additional inconsistencies introduced at the application layer. Heath et al. [21] analyse disappearing message features across WhatsApp, Telegram, and Snapchat, showing that expired content commonly persists in cache directories, quoted-message structures, forwarded-message logs, notification artefacts, and cloud backups. These findings demonstrate that ephemerality is often enforced at the user-interface or protocol level rather than through comprehensive suppression of underlying artefacts, leaving multiple residual exposure paths.

User perception further complicates this picture. Schnitzler et al. [22] show that users frequently conflate logical deletion with secure erasure, and often assume that disappearing messages eliminate all traces by default. These assumptions can expand the effective threat surface, as users may rely on ephemerality for sensitive communication without appreciating the fact that data may persist elsewhere.

This body of work imposes strict boundaries on what Ember can plausibly claim. Ember's design deliberately minimises plaintext exposure by avoiding persistent message databases, enforcing ciphertext-only storage, aggressively clearing in-app state, and applying TTL-driven deletion of message records. This removes a substantial class of application-layer artefacts, particularly those arising from long-lived plaintext stores, message indexing structures, and notification previews. However, Ember cannot guarantee secure deletion at the level of the Android filesystem. As the cited literature demonstrates, deleted data may persist in journaling regions, write-ahead logs, cached metadata, or untrimmed flash storage, and may be recoverable by a forensically capable adversary with physical device access.

Accordingly, Ember explicitly constrains its threat model. TTL-based deletion ensures that expired messages become inaccessible through the Ember application interface, but does not assert physical sanitisation of underlying storage. Full-disk encryption and hardware-backed keystore protections mitigate some risks by binding decryption capability to user authentication and trusted execution environments. Nevertheless, ephemerality on Android must be framed as limiting live exposure and post-compromise readability *within the application*, rather than as preventing all forms of forensic recovery under device compromise. Ember's evaluation makes this boundary concrete by identifying a specific case. This case is the retention of per-message keys in a dedicated key store after ciphertext deletion, where application-level minimisation is currently incomplete.

2.7 Mobile Implementation Reality: Push Notifications, Metadata Leakage, and Android Key Storage

While Ember avoids reliance on central servers for message relay, mobile operating systems impose architectural constraints that materially affect privacy and key-management guarantees. Two aspects are particularly relevant: metadata leakage via push notification infrastructures, and the practical realities of secure key storage on Android devices. Both represent adversarial surfaces that persist even when transport- and protocol-level security is strong.

Push Notifications as a Metadata Leakage Vector.

Recent empirical work shows that push notification infrastructures frequently undermine the privacy guarantees of otherwise secure messaging systems. Samarin et al. [23] analyse 21 popular messaging applications and demonstrate that more than half transmit sensitive metadata to third-party push providers such as Google Firebase Cloud Messaging (FCM), often in plaintext. Leaked information includes user identifiers, sender and recipient details, and in some cases message content itself. Importantly, this leakage arises not from cryptographic weaknesses in end-to-end encryption protocols, but from application-layer notification payloads routed through external infrastructure.

The study highlights a systemic issue: developer-facing push notification APIs encourage embedding rich payload data, while end-to-end encryption at the notification layer is left as an optional and frequently omitted safeguard. As a result, even applications that correctly implement strong cryptographic protocols may inadvertently expose metadata through push delivery channels. These findings are reinforced by documented legal processes in which push tokens and associated metadata have been obtained from platform providers.

For Ember, this literature motivates a deliberate avoidance of third-party push notification services for message delivery. By relying on a serverless peer-to-peer overlay rather than FCM or APNs, Ember eliminates a well-documented class of metadata leakage pathways. This design choice trades OS-native wake-up mechanisms for tighter control over message transport and metadata exposure, making availability and delivery explicit responsibilities of the application rather than outsourced infrastructure.

Android Key Storage: Empirical Use and Misuse.

A second constraint concerns secure key storage on Android. While the platform exposes trusted hardware through the Keystore API, empirical evidence shows that its protections are inconsistently used in practice. Blessing et al. [24] present a large-scale measurement study analysing hundreds of thousands of Android applications and find widespread misuse or avoidance of hardware-backed key storage. A majority of apps handling sensitive data do not use trusted hardware at all, and only a small fraction leverage the StrongBox secure element. Even among applications that invoke the Keystore API, insecure parameter choices and delegation of key management to third-party libraries are common.

The study further shows that performance and latency considerations strongly influence developer behaviour. Hardware-backed operations, particularly those involving StrongBox, introduce measurable overheads that discourage their use in latency-sensitive paths. The analysis also clarifies a key limitation of Android's hardware-backed keystores: while they effectively prevent key exfiltration, they do not prevent an attacker with sufficient privileges from invoking those keys on-device.

These findings inform Ember's key-management strategy. Ember stores long-term Curve25519 identity material via the Android Keystore where supported, but does not rely on StrongBox or device-level guarantees as a universal security baseline. Ephemeral ratchet keys including, root keys, chain keys, and per-message keys, are derived on demand and are intended to be short-lived; as discussed in the evaluation, ensuring that they are actually discarded once consumed is the locus of the residual forward-secrecy gap that this paper identifies and proposes to close.

Implications for Ember.

This body of work reinforces two architectural positions adopted by Ember. First, metadata minimisation is achieved primarily by eliminating dependence on third-party push infrastructures rather than attempting to retrofit encryption onto inherently leaky delivery channels. Second, hardware-backed key storage is treated as a defence-in-depth mechanism rather than a foundational assumption. Ember pairs opportunistic use of trusted hardware for long-term identity keys with protocol- and application-level minimisation strategies for ephemeral ratchet state, ensuring that confidentiality does not hinge solely on the correctness or availability of platform-specific security features.

The broader implication is that secure messaging on mobile platforms is shaped as much by operating system behaviour and developer-facing APIs as by cryptographic protocol design. Ember adopts a conservative posture within this environment by reducing reliance on external infrastructures, minimising retained state, and making its security guarantees explicit and narrowly scoped.

3 Threat Model and Assumptions

This section defines the adversary model, trust assumptions, security goals, and explicit non-goals for Ember. The intent is to make Ember's security claims evaluable against established threat models while avoiding overstated guarantees. Following prior SoK work, Ember distinguishes between *content security* (confidentiality/integrity/authenticity of message payloads) and *metadata exposure* (who talks to whom, when, and how often), and treats mobile platform realities and retained state as first-class constraints [3,23].

3.1 Adversary Capabilities

The authors assume the following adversarial capabilities.

Network observation (passive).

An adversary may observe traffic on one or more network segments, including local Wi-Fi, upstream ISPs, or overlay-adjacent vantage points. Ember operates over an encrypted IPv6 overlay (Yggdrasil), which provides link encryption and key-derived addressing at the routing layer; however, Ember does not assume this yields anonymity or traffic-analysis resistance. An observer with visibility into overlay entry/exit points may still infer communication timing and peer relationships through correlation, even if payloads are encrypted [17,25]. The dynamic evaluation in Section 7 characterises precisely what such an observer sees from a relay-adjacent vantage point.

Active network interference.

An adversary may drop, delay, replay, and reorder packets; inject malformed frames; attempt downgrade or protocol-confusion attacks; and perform denial-of-service (DoS) against the transport layer (e.g., repeated connection attempts, resource exhaustion against the listener). Ember's application-layer envelope includes authenticated ciphertext (AES-GCM), an explicit HMAC gate keyed from the current ratchet root key, and per-message Double Ratchet state; consequently, successful payload tampering or state injection should be detected before any ratchet advance or decryption, but availability is not guaranteed under sustained DoS. Message loss, reordering, and duplication are tolerated by the Double Ratchet's skipped-message-key handling rather than by a delivery service.

Malicious peers and impersonation attempts.

A peer contact may behave arbitrarily: send malformed envelopes, attempt state desynchronisation, replay or reorder messages, or attempt to impersonate a different sender identity. Ember assumes contacts are not inherently trusted. Identity is anchored to long-term Curve25519 identity keys, surfaced to the user as key fingerprints, established through the X3DH handshake, and confirmed via a pragmatic trust-on-first-use

(TOFU) workflow with optional manual fingerprint verification. Because Ember now deploys X3DH and the Double Ratchet, it provides per-message forward secrecy and post-compromise security under the standard Signal-style assumptions; it does not, however, claim formally verified resistance to all key-compromise-impersonation variants, and it prioritises correct authenticated encryption and explicit sender-binding at the envelope layer in addition to the ratchet's guarantees.

Bounded device compromise.

The authors assume *bounded* endpoint compromise scenarios:

- **Opportunistic compromise:** malware or a local attacker gains access to the application sandbox, obtains the encrypted database, or reads screen content while the device is unlocked.
- **Privilege escalation/root:** a stronger attacker gains root or debugging capability, enabling memory inspection, IPC manipulation, and direct interaction with the app process.

Under root compromise, hardware-backed keystore protections may prevent key *exfiltration* but do not necessarily prevent key *use* on-device (i.e., an attacker can often drive cryptographic operations through the compromised process) [24]. Ember therefore treats root compromise as a severe condition that can break confidentiality for currently accessible sessions and any plaintext visible in memory. The Double Ratchet limits the temporal scope of such compromise through post-compromise security provided ephemeral key state is actually discarded once consumed. The current implementation's retention of per-message keys after ciphertext deletion (Section 7.5.6) is therefore a material qualifier on the strength of this property against a *forensic* adversary who later images the device.

Physical access and forensic recovery.

A forensically capable adversary may obtain a device image or access residual storage artefacts. Ember enforces TTL deletion and avoids plaintext persistence in the database, but the literature shows that Android deletion semantics do not imply physical erasure: artefacts can persist in journaling structures, WAL regions, and unallocated blocks [19,21]. In addition, Ember currently retains per-message key state beyond ciphertext deletion, which expands the forensic recovery surface for expired messages. Ember explicitly limits its claims accordingly (see Non-Goals and Section 7.5.6).

3.2 Trust Assumptions

Ember's security properties depend on explicit assumptions about endpoints, key storage, and cryptographic primitives.

Endpoint computing base.

Ember assumes the endpoint OS and runtime are sufficiently trustworthy that:

- the app's process and memory are not continuously instrumented by an attacker, and
- the UI is not systematically subverted (e.g., full accessibility abuse, screen overlay attacks) during sensitive operations such as fingerprint verification.

This is a pragmatic assumption shared by most mobile secure messaging designs: if the endpoint is fully compromised, protocol-level guarantees degrade sharply regardless of transport properties [3].

Android Keystore guarantees (opportunistic hardware backing).

Ember assumes Android Keystore behaves as documented for generating and protecting long-term secret material, with hardware-backed storage (TEE) available on many devices. However, Ember does *not* assume StrongBox availability or performance suitability, and does not rely on StrongBox for latency-sensitive operations. This aligns with empirical observations that secure hardware usage is fragmented and frequently avoided due to performance and integration constraints [24].

Cryptographic primitive security.

Ember assumes standard cryptographic primitives are secure and correctly implemented: Curve25519 for X3DH and Double Ratchet Diffie–Hellman operations, AES-256-GCM for confidentiality/integrity, HMAC-SHA256 for explicit integrity gating and sender-binding, HKDF-SHA256 for all ratchet key derivation (root, sending, and receiving chains and the envelope HMAC key), and SHA-256 for fingerprints and identifiers. Ember also assumes a cryptographically secure RNG for nonce generation and key generation. Under these assumptions, message confidentiality, integrity, and the ratchet’s forward secrecy and post-compromise security reduce to correct key management and correct use of the underlying primitives, rather than relying on the overlay network for security.

Overlay transport as reachability, not anonymity.

Ember treats Yggdrasil as a reachability and encrypted-link substrate that simplifies peer connectivity via stable key-derived IPv6 addressing and encrypted links. Ember does not treat the overlay as providing sender/receiver anonymity, unlinkability, cover traffic, or resistance to global traffic analysis; these remain out of scope without additional anonymity mechanisms [17,25]. The fixed-size, fixed-cadence link-layer profile observed empirically in [Section 7](#) is treated as an incidental obfuscation against a relay-adjacent observer, not as a relied-upon guarantee.

3.3 Security Goals

Ember aims to satisfy the following goals within the above assumptions and constraints.

Confidentiality of message content.

Only intended endpoints should be able to decrypt message payloads. Ember enforces this via application-layer authenticated encryption (AES-256-GCM) using per-message keys derived from the Double Ratchet, with plaintext never persisted to disk and decrypted only in memory for rendering. Ciphertext-only persistence is intended to minimise risk from device seizure or sandbox-level data exfiltration.

Per-message forward secrecy and post-compromise security.

Compromise of cryptographic state at one point in time should not retrospectively expose past messages (forward secrecy), and the protocol should heal after a transient compromise such that future messages regain confidentiality once the adversary loses access (post-compromise security). Ember pursues both via the Double Ratchet: a Diffie–Hellman ratchet step is performed on every direction reversal, advancing the root key and deriving fresh chain keys, while same-direction messages advance a symmetric KDF chain. This goal is the principal advance over the static-key design previously described, and [Section 7](#) provides runtime evidence that the ratchet is in fact ratcheting. The goal is qualified in the current build by the retention of per-message key state after ciphertext deletion, which weakens forward secrecy specifically against a forensic adversary; closing this gap is the highest-priority remediation item.

Integrity and authenticity of messages.

Recipients should detect tampering and reject forged messages. Ember’s envelope processing uses an explicit HMAC-SHA256 verification step prior to any ratchet advance or decryption (verify-then-decrypt), keyed from the current ratchet root key, and binds sender metadata and the per-message ratchet header fields into the authenticated material. This goal is aligned with baseline secure messaging objectives under an active network adversary [3].

Data minimisation and bounded retention.

Ember aims to minimise retained sensitive data by:

- storing only ciphertext at rest,
- enforcing TTL expiration through WorkManager and targeted delayed cleanup jobs, with empirically tight deletion latency, and
- avoiding push-notification payloads that disclose content or sensitive identifiers.

This is motivated both by secure messaging threat models that treat metadata and retained artefacts as primary risks, and by empirical evidence that “secure” messengers often leak sensitive data through auxiliary infrastructures [23]. Ember additionally aims to minimise retained per-message key state symmetrically with ciphertext deletion.

Practical key lifecycle management.

Ember targets a key lifecycle that is implementable on Android without a coordinating server:

- long-term Curve25519 identity keys protected via Keystore-backed storage where available,
- X3DH initial key agreement using long-term identity, signed pre-key, and (where available) one-time pre-key contributions, with pre-key bundles exchanged directly between peers,
- Double Ratchet per-message key evolution providing FS and PCS without server coordination,
- explicit user-verifiable identity fingerprints to support pragmatic trust establishment.

Availability under realistic constraints.

Ember aims for usable delivery and reception when peers are reachable on the overlay, but it does not claim high availability under adversarial DoS, mobile background execution limits, or prolonged peer offline periods. The system explicitly prefers minimised infrastructure over centralised reliability guarantees, consistent with its serverless design constraints.

3.4 Explicit Non-Goals

Ember intentionally does *not* claim the following properties in its current design.

Anonymity and strong metadata privacy.

Ember does not claim sender/receiver anonymity, unobservability, or robust resistance to traffic analysis against a global passive adversary. Overlay routing may reduce central collection points, but timing and relationship inference remain plausible under realistic observation models [25]. Any stronger metadata-hiding properties would require additional mechanisms (e.g., cover traffic, mixing, rendezvous privacy), which are out of scope for the present implementation.

Unqualified forward secrecy against a forensic adversary.

Although Ember implements the Double Ratchet and therefore provides per-message forward secrecy at the protocol level, the current build does not symmetrically purge per-message key material from its dedicated key store when the corresponding ciphertext is deleted on TTL expiry. Consequently, against a forensic adversary who later images the device, expired messages may remain recoverable using retained keys. Ember therefore does not claim *unqualified* forward secrecy against device imaging until this gap is closed; the protocol-level FS/PCS guarantees against a transient online compromise are unaffected (Section 7.5.6).

End-to-end empirical validation of the X3DH handshake.

The dynamic evaluation in this paper does not exercise an X3DH handshake within its capture window, because the test conversation pre-existed the session. The X3DH handshake is therefore described from the

implementation and is supported by the observation of routine pre-key bundle republication, but a fresh end-to-end handshake capture is deferred to future work and is not claimed here.

Secure deletion against forensic adversaries.

TTL expiry and database cleanup ensure that expired messages become inaccessible through Ember and are removed from application-level ciphertext stores. Ember does not claim physical secure deletion on Android storage media, nor does it claim elimination of all artefacts recoverable via filesystem journaling, WAL remnants, or raw block carving [19,20]. Ephemerality is therefore defined as *application-level access minimisation*, not guaranteed sanitisation.

Group messaging and multi-device consistency.

Ember currently targets one-to-one communication. It does not claim group messaging security properties (e.g., MLS-style epoch-based membership updates, scalable FS/PCS under group churn), nor does it claim multi-device state consistency. Extending Ember to groups would require explicit re-evaluation of authentication services, delivery semantics, and group state management assumptions [10].

Comprehensive endpoint compromise resistance.

Ember does not claim to protect message confidentiality if an attacker has sustained control of an unlocked device, captures UI content via overlays, or compromises the app process at runtime. As with most messengers, endpoint compromise is treated as a dominant failure mode; Ember's mitigations focus on reducing the value of stored artefacts, limiting retention, and bounding the temporal window of compromise, not on defeating a fully compromised client.

4 System Overview

Ember is an Android-first, peer-to-peer secure messenger that operates over the Yggdrasil IPv6 overlay network without centralised relay infrastructure. The system is intentionally conservative in its protocol surface area: it prioritises message confidentiality, integrity, and per-message compromise resilience, minimises retained plaintext, and constrains feature scope to keep security properties analysable and implementation risk bounded.

4.1 High-Level Architecture

Ember follows a layered architecture that separates user interface logic, domain orchestration, cryptographic operations, persistence, and network transport. This separation is deliberate: it reduces the risk of cross-layer security coupling, improves testability, and enables reviewers to reason about security-critical paths without conflating UI behaviour with cryptographic correctness.

UI layer (presentation and interaction).

The UI is implemented using Jetpack Compose and a ViewModel-driven state model. ViewModels expose immutable UI state, mediate user actions (send message, add contact, verify fingerprint), and subscribe to reactive streams of conversation and network status. The UI layer contains no cryptographic primitives and does not access raw storage directly; it delegates all security-sensitive operations to the domain layer.

Domain layer (orchestration and state machines).

The domain layer centres on a repository-style orchestration component (`ChatRepository`). This layer coordinates: (i) X3DH key agreement and Double Ratchet evolution, (ii) authenticated encryption and HMAC gating, (iii) persistence and retrieval of message records, and (iv) transport send/receive flows. The

repository enforces security invariants, including *verify-then-decrypt* processing, plaintext non-persistence, TTL scheduling, and correct ratchet-state advancement on direction reversal.

Cryptographic layer (encryption, authentication, key management).

Ember's cryptographic core implements an X3DH initial key agreement followed by a Double Ratchet protocol for all subsequent message encryption, broadly following the structure used in production by the Signal protocol family. The X3DH handshake establishes a shared root key from long-term identity, signed pre-key, and (where available) one-time pre-key Curve25519 contributions; the resulting root key is consumed by the Double Ratchet to derive the initial sending and receiving chain keys. Message payloads are then encrypted with AES-256-GCM under per-message keys derived from the ratchet chains, using a fresh 12-byte random nonce per message. In addition, Ember computes an explicit HMAC-SHA256 over envelope components, including the per-message ratchet header fields (`dhPub`, `msgIdx`, `pn`) and sender-binding fields, keyed from the current ratchet root key, and verifies the HMAC prior to any ratchet-state update or decryption. Long-term identity keys are protected using Android Keystore-backed mechanisms where available, and user-visible fingerprints are derived using SHA-256 over the identity public key to support out-of-band verification.

Storage layer (encrypted persistence with plaintext minimisation).

Ember persists application state using Room with SQLCipher-backed encrypted SQLite storage. The storage model is structured around contacts, conversations, and message records, alongside a dedicated key store holding per-message ratchet key material. The critical storage invariant is that **only ciphertext is stored** for message bodies; plaintext is computed only in-memory for rendering and is not written to disk. Messages are associated with TTL policies and an `expiresAt` timestamp, enabling automated deletion of expired records. As discussed in [Section 7.5.6](#), the current build deletes expired ciphertext records but does not yet symmetrically purge the corresponding per-message keys from the key store; closing this gap is a targeted remediation item. Database schema changes are managed through explicit migrations to avoid destructive upgrade paths.

Network layer (serverless transport over Yggdrasil).

Ember uses direct TCP connections between peers on Yggdrasil IPv6 addresses. Transport is realised via: (i) an outbound client that establishes TCP connections and sends framed message envelopes, and (ii) an inbound foreground listener service that accepts connections and parses envelopes. A lightweight, length-prefixed framing format is used to delimit JSON-serialised envelopes over TCP. A bridging component forwards received envelopes from the Android service context to the dependency-injected domain layer, where integrity verification, ratchet advancement, and decryption occur.

End-to-end message flow.

The end-to-end pipeline is designed so that authentication occurs before any cryptographic state is advanced or any plaintext is produced:

1. On send, the UI requests a send operation from the domain layer.
2. The domain layer derives the next message key from the sending chain (performing a Diffie–Hellman ratchet step if this is the first message after a direction reversal), encrypts plaintext into ciphertext (AES-GCM), computes a HMAC over the envelope fields (including `dhPub`, `msgIdx`, and `pn`), and persists the ciphertext record.
3. The network layer transmits a framed envelope containing ciphertext, nonce, ratchet header fields, and authentication data.
4. On receive, the listener parses an envelope and forwards it to the domain layer.

5. The domain layer verifies the HMAC (*verify-then-decrypt*); only on success does it detect whether the remote DH public key has changed, perform a DH ratchet step if so, derive the appropriate receiving-chain message key, decrypt the ciphertext, and emit plaintext only to the in-memory UI projection.
6. TTL metadata is used to schedule expiration cleanup for stored ciphertext.

This ordering is intentional: it ensures that malformed or unauthenticated envelopes are rejected without producing plaintext *and without advancing any cryptographic state*, and that stored message content remains encrypted at rest.

4.2 Design Principles

Ember's design is guided by a small set of explicit principles that trade breadth of features for reduced trusted infrastructure and improved analysability.

Serverlessness and reduced infrastructure trust.

Ember is designed to operate without central servers for identity, message relay, account provisioning, or pre-key distribution. Peers communicate directly over the overlay using Yggdrasil IPv6 addresses and a configurable port, and exchange X3DH pre-key bundles directly rather than through a global directory. This reduces reliance on third-party relays and avoids the typical “provider as metadata aggregation point” characteristic of centralised messaging platforms. The system explicitly treats the overlay as a reachability substrate rather than an anonymity system; Ember does not claim comprehensive metadata protection beyond minimising retained application-level data.

Minimal retained plaintext.

A core objective is to minimise plaintext persistence and reduce forensic value of stored state. Ember enforces ciphertext-only message storage and performs decryption only transiently for user display. Notifications are privacy-first and do not disclose message content. TTL policies limit how long ciphertext records remain in the local database, reducing long-term retention by default.

Fail-closed processing and defensive parsing.

Ember's message processing pipeline is designed to fail closed. Envelopes are authenticated before any ratchet advance or decryption, and malformed inputs are rejected early. This principle aims to bound the impact of active network adversaries and malicious peers by making “unauthenticated state advancement or plaintext production” structurally difficult.

Strong compromise resilience with a constrained surface.

Ember adopts the Signal-style X3DH-plus-Double-Ratchet design to obtain per-message forward secrecy and post-compromise security, but deliberately keeps the surrounding surface small: a single deterministic conversation identity, an explicit HMAC gate, a single KDF (HKDF-SHA256) used throughout the ratchet with domain-separated parameters, and no group or multi-device state. This keeps the composed protocol (X3DH root key, ratchet evolution, envelope authentication) tractable to reason about and test, in line with the compositional concerns raised by the formal ratcheting literature [7].

Pragmatic key lifecycle management.

Rather than assuming ideal hardware security, Ember uses platform keystore protections opportunistically for long-term identity keys and pairs them with protocol-level minimisation of ephemeral ratchet state. User-visible fingerprints provide a human-verifiable trust anchor to support manual verification and mitigate silent substitution risks in adversarial environments.

4.3 Functional Scope

Ember's implemented functionality is intentionally constrained to a clear baseline set of capabilities. The scope below reflects the current system behaviour rather than aspirational features.

Implemented capabilities.

- **One-to-one (1:1) messaging:** direct peer messaging between two endpoints, with conversation identifiers deterministically derived to provide stable mapping between contacts and threads.
- **X3DH initial key agreement:** establishment of a shared root key from long-term identity, signed pre-key, and (where available) one-time pre-key Curve25519 contributions, with pre-key bundles exchanged directly between peers.
- **Double Ratchet per-message key evolution:** a Diffie–Hellman ratchet step on each direction reversal and symmetric KDF-chain advances within unidirectional bursts, providing per-message forward secrecy and post-compromise security without server coordination.
- **End-to-end payload protection:** AES-256-GCM encryption per message using per-message keys derived from the ratchet, with explicit HMAC-SHA256 verification prior to any ratchet advance or decryption.
- **Encrypted local persistence:** SQLCipher-encrypted database via Room, storing message ciphertext and metadata while ensuring plaintext is never written to disk.
- **TTL expiration and background cleanup:** messages carry TTL metadata and an expiration timestamp; WorkManager-based periodic sweeps complemented by targeted delayed jobs remove expired ciphertext records, with empirically tight (~100 ms) deletion latency.
- **Identity fingerprinting and verification support:** SHA-256-derived fingerprints over identity public keys rendered for user comparison to support pragmatic trust establishment and verification workflows.
- **Privacy-first notifications:** notifications avoid displaying message content and use conservative lock-screen visibility settings to reduce inadvertent disclosure.
- **Serverless transport over Yggdrasil:** direct TCP send/receive over overlay IPv6 addresses, including an inbound listener running as a foreground service for reliable reception.
- **Operational diagnostics:** UI-accessible network and status views intended to support troubleshooting and to make transport state visible to the user.

Explicit exclusions in the current system.

- **Group messaging:** not implemented; no MLS-style group state or sender-key semantics are provided.
- **Multi-device operation:** a user's identity, ratchet state, and message history are bound to a single device; no cross-device synchronisation is provided.
- **Asynchronous offline delivery:** there is no store-and-forward capability; both peers must be reachable for delivery to succeed.
- **Centralised identity services:** Ember does not provide accounts, phone-number identity binding, or a global directory; contacts are created through direct endpoint configuration, pre-key exchange, and verification workflows.
- **Symmetric purge of per-message keys on TTL expiry:** the current build deletes expired ciphertext but retains the corresponding per-message keys in the key store ([Section 7.5.6](#)); this is an identified remediation item rather than a designed-in property.

These choices position Ember as a deliberately minimal secure messenger: it provides strong content confidentiality, integrity, and per-message compromise resilience within a serverless peer-to-peer environment, while keeping protocol complexity low enough that the system's guarantees, failure modes, and trust dependencies remain explicit and reviewable.

4.4 Network Architecture

Ember's network architecture prioritises direct, serverless reachability and protocol simplicity over the abstraction and reliability guarantees typically offered by centralised messaging infrastructure.

4.4.1 Transport Model

Ember uses direct peer-to-peer TCP communication over an encrypted IPv6 overlay network. Each peer is addressed using a stable IPv6 address derived from cryptographic key material by the overlay, and listens on a configurable TCP port. This addressing model provides globally unique, collision-resistant endpoints without requiring DNS, NAT traversal services, or central rendezvous servers.

TCP is used as the transport protocol for three primary reasons. First, it provides reliable, ordered delivery semantics that simplify message handling at the application layer. Second, it allows Ember to avoid implementing its own congestion control, retransmission logic, or fragmentation scheme. Third, TCP is well-supported by Android's networking APIs and integrates cleanly with long-lived background services when required.

The overlay network provides encrypted links between nodes, but Ember does not treat the overlay as a security boundary. All application-level security properties, including confidentiality, integrity, sender authentication, and the Double Ratchet's forward secrecy and post-compromise security, are enforced independently at the message layer. As a result, Ember remains robust even if the overlay is misconfigured, partially observable, or compromised at the routing level. The overlay is therefore treated strictly as a reachability substrate rather than a trust anchor.

Communication is symmetric: every node can both initiate outbound connections and accept inbound connections. There is no concept of a relay or broker within Ember itself. Message delivery succeeds only when peers are simultaneously reachable on the overlay and listening on the expected port, reinforcing Ember's peer-to-peer and availability-bounded design philosophy.

4.4.2 Envelope and Framing Protocol

Messages are transmitted as discrete envelopes over TCP using a simple length-prefixed framing protocol. Framing is required because TCP is a byte-stream protocol with no inherent message boundaries.

Each transmitted message consists of:

1. A fixed-size 4-byte big-endian unsigned integer indicating the length of the envelope payload in bytes.
2. A JSON-serialised envelope of the specified length.

The length prefix allows the receiver to deterministically read exactly one envelope from the stream, reject oversized or malformed payloads early, and avoid ambiguity arising from partial reads or stream concatenation. This framing strategy is intentionally minimal and avoids complex multiplexing or stream management logic.

The envelope itself contains all metadata required for message processing, ratchet advancement, and verification. Beyond the routing and lifecycle fields, the envelope carries the per-message Double Ratchet state required by the receiver to derive the corresponding message key. Typical fields include:

- protocol version identifier,
- conversation identifier,
- sender identifier or name,
- timestamp and TTL metadata,
- message type (e.g., user message, pre-key bundle publication),

- the sender's current Double Ratchet DH public key (`dhPub`, Base64),
- the index of this message within the current sending chain (`msgIdx`),
- the length of the previous sending chain (`pn`), supporting chain skipping,
- ciphertext (Base64-encoded),
- nonce (Base64-encoded),
- HMAC (Base64-encoded).

The `dhPub` field allows the receiver to detect direction reversals and trigger a Diffie–Hellman ratchet step when the value differs from the previously observed remote DH public key. The `msgIdx` field identifies the message's position within the sender's current sending chain, while `pn` records the length of the sender's previous sending chain so that the receiver can correctly skip and retain keys for any out-of-order messages from that prior chain. These fields are protected by the envelope HMAC and are processed only after authentication has succeeded.

No plaintext message content is ever included in the envelope. Metadata fields are intentionally constrained to those required for routing, authentication, ratchet advancement, and lifecycle management. In particular, Ember avoids embedding user profile data, contact lists, or message previews in envelopes, reducing the risk of metadata amplification at the transport layer.

On reception, envelopes are parsed and validated defensively. Length mismatches, malformed JSON, missing required fields, or unsupported protocol versions result in immediate rejection. Crucially, cryptographic verification (HMAC validation) is performed before any ratchet state update or decryption attempt. This ordering enforces a fail-closed processing model in which unauthenticated data cannot advance cryptographic state or cause plaintext to be produced or persisted.

4.4.3 Connection Management

Connection management in Ember is intentionally simple and conservative. Each outbound message send establishes a TCP connection to the peer's IPv6 address and port using a bounded retry strategy. Long-lived connections are not assumed to be stable, reflecting the realities of mobile networking, background execution limits, and dynamic overlay routing.

Outbound connections use explicit timeouts to bound resource consumption:

- connection establishment timeout,
- read timeout for framed envelope reception,
- write timeout for outbound payload transmission.

If a connection attempt fails, Ember retries using exponential backoff up to a fixed maximum number of attempts. This limits the impact of transient network failures while avoiding unbounded retry loops that could drain battery or create denial-of-service conditions.

Inbound connections are accepted by a dedicated listener service (discussed below) and processed sequentially. Ember does not attempt to multiplex multiple concurrent message streams over a single connection, nor does it maintain per-peer connection pools. This choice reduces state complexity and failure modes at the cost of some efficiency, which is acceptable given Ember's low expected message throughput and emphasis on analysability. As the dynamic evaluation shows, this per-connection model makes TCP connection establishment the dominant contributor to end-to-end latency, motivating connection pooling as future work.

Connection state is modelled explicitly and exposed to the UI via reactive state streams. States such as `DISCONNECTED`, `LISTENING`, `CONNECTED`, and `ERROR` allow both users and developers to

observe network behaviour directly. This transparency is intentional: rather than hiding connectivity issues behind opaque retry logic, Ember surfaces transport state to support debugging, user understanding, and security evaluation.

4.4.4 Background Receive Capability

Receiving inbound connections on Android imposes strict constraints due to modern background execution limits. Android aggressively restricts background network access and process lifetime in order to preserve battery life, making traditional always-on listeners infeasible for most applications.

To ensure reliable inbound message reception, Ember employs a foreground service that hosts the TCP listener. Running as a foreground service grants the application elevated execution priority and permits continuous socket listening while the device is idle or the UI is not in the foreground. This design choice is explicit and user-visible: the service presents a persistent notification indicating that Ember is actively listening for incoming connections.

The use of a foreground service is justified on functional and security grounds. Functionally, it is the only reliable mechanism to support serverless peer-to-peer reception without relying on third-party push notification infrastructure. From a security perspective, it avoids the need to route message metadata or wake-up signals through external providers, which have been shown to introduce significant privacy leakage in mainstream messaging systems [23].

However, this approach comes with clear trade-offs. Foreground services consume more system resources, may impact battery life, and require explicit user consent. Ember treats these costs as acceptable within its threat model and design goals, preferring transparent, user-controlled availability over opaque reliance on centralised notification channels.

In summary, Ember's network architecture is deliberately straightforward: direct TCP over cryptographically addressed IPv6 peers, minimal framing carrying the per-message ratchet header, conservative connection management, and explicit foreground execution for receive capability. This simplicity reduces the attack surface, makes failure modes explicit, and aligns with Ember's broader goal of providing secure peer-to-peer messaging without hidden infrastructure dependencies.

5 Cryptographic Design

Ember's cryptographic design adopts the Signal-style X3DH-plus-Double-Ratchet structure to obtain per-message forward secrecy and post-compromise security, while deliberately keeping the surrounding protocol surface small and explicitly analysable. The system enforces strong confidentiality and integrity for message payloads and evolves key material on a per-message basis without any central key distribution service.

5.1 Conversation Identity and Keying

Each Ember conversation is defined as a one-to-one relationship between two peer endpoints. Conversations are identified using a deterministic conversation identifier derived from both peers' addressing material. The identifier is computed as a UUID v3 hash over a canonicalised, sorted representation of the two endpoints (e.g., IPv6 address and port), ensuring that:

- the same pair of peers always derives the same conversation identifier,
- conversation identifiers are collision-resistant under standard hash assumptions,
- no external coordination or server-side allocation is required.

Each device generates a long-term Curve25519 identity key pair at first launch and protects the private key via the Android Keystore where available. This identity key is the root of trust for the conversation: it anchors the X3DH handshake and is the basis for the user-visible fingerprint. Per conversation, Ember maintains evolving Double Ratchet state for the root key, the sending chain key, the receiving chain key, the local and remote DH ratchet key pairs/values, and a set of retained per-message keys for skipped or recently consumed messages. Compromise of one conversation's ratchet state does not affect other conversations, even with the same peer.

5.2 Initial Key Agreement (X3DH)

Ember establishes the initial shared secret for a conversation using an Extended Triple Diffie–Hellman (X3DH) handshake over Curve25519, following the structure used in the Signal protocol family and analysed by Hashimoto et al. [8]. Each peer publishes a pre-key bundle comprising its long-term identity key, a signed pre-key, and a set of one-time pre-keys. Because Ember is serverless, pre-key bundles are exchanged directly between peers (carried in a dedicated `PREKEY_BUNDLE` envelope type) rather than being held in a global directory.

The X3DH computation combines several Diffie–Hellman results; between the initiator's identity key and the responder's signed pre-key, between the initiator's ephemeral key and the responder's identity key, between the initiator's ephemeral key and the responder's signed pre-key, and (where a one-time pre-key is available) between the initiator's ephemeral key and that one-time pre-key. The concatenated DH outputs are passed through HKDF-SHA256 to derive the initial root key, which is then handed to the Double Ratchet to seed the first sending and receiving chains. Use of a one-time pre-key, where available, strengthens forward secrecy for the initial messages by ensuring that the corresponding ephemeral private key is consumed and discarded.

The handshake binds the established secret to long-term identities, providing implicit mutual authentication subject to correct verification of identity keys; Ember surfaces this to the user via fingerprints and a TOFU workflow (Section 5.7). As noted in the threat model and evaluation, the dynamic test in this paper does not capture a fresh X3DH handshake end-to-end, because the test conversation pre-existed the capture window; the handshake is therefore described from the implementation, with end-to-end empirical validation deferred to future work.

5.3 Per-Message Key Evolution (Double Ratchet)

After the X3DH handshake seeds the root key, Ember evolves keys on a per-message basis using the Double Ratchet algorithm. Two complementary mechanisms operate together.

Diffie–Hellman ratchet (on direction reversal).

Whenever the direction of communication reverses, that is, when a peer receives a message whose `dhPub` differs from the previously observed remote DH public key, the receiver performs a DH ratchet step. A new Diffie–Hellman shared secret is derived from the remote sender's freshly rotated DH public key and the receiver's local DH private key; the root key is advanced through HKDF-SHA256; a new receiving chain key is derived; the previous sending chain is discarded; and a fresh local sending DH key pair is generated for the receiver's next outbound message. This is the source of post-compromise security: an adversary who has learned the current state cannot derive keys for messages following a future DH ratchet step without also compromising the new DH private key.

Symmetric KDF chain (within a unidirectional burst).

Within a sequence of messages sent in the same direction, no DH operation is performed. Instead, each successive message key is derived by advancing a symmetric KDF chain via HKDF-SHA256. This provides forward secrecy at low computational cost: once a chain key has been advanced, prior message keys cannot be recomputed from the new chain key. Each message is encrypted under a one-shot key derived from the chain, so the structural risk of nonce reuse under a fixed key is bounded even in the presence of randomness failures, because a repeated nonce would have to coincide with a repeated derived message key for any GCM security failure to occur.

Chain accounting and out-of-order delivery.

The `msgIdx` and `pn` envelope fields carry the message's position in the current sending chain and the length of the previous sending chain, respectively. Together they allow the receiver to skip and retain keys for out-of-order or delayed messages from a prior chain without losing the ability to decrypt them when they later arrive. Retained per-message keys are stored in a dedicated key store; as discussed in [Section 7.5.6](#), ensuring these retained keys are purged once they are no longer needed (and in particular when the corresponding ciphertext expires) is essential to realising the forward-secrecy property in full.

HKDF-SHA256 is used as the sole KDF throughout the ratchet, with domain-separated salt and info parameters distinguishing root-key updates from chain-key advances and from derivation of the envelope HMAC key. This construction ensures domain separation between successive ratchet states and prevents cross-context key reuse.

5.4 Confidentiality Mechanism

Message confidentiality is provided using AES-256-GCM, an authenticated encryption with associated data (AEAD) construction that provides confidentiality and integrity under standard cryptographic assumptions. Ember encrypts each message under a fresh 256-bit per-message key derived from the Double Ratchet, and generates a fresh, cryptographically secure random 12-byte nonce for each message.

Nonce generation follows a strict uniqueness requirement: a 12-byte nonce is generated using a cryptographically secure random number generator for every encryption operation. Nonces are never reused with the same key. Because each message key is itself a one-shot value produced by the ratchet, the design avoids catastrophic nonce-reuse failure modes associated with GCM and does not rely on counters or shared nonce-state synchronisation between peers.

AES-GCM is used solely for encrypting message payloads. Plaintext exists only transiently in memory during encryption and decryption operations and is never persisted to disk. Ciphertext, nonce, ratchet header fields, and authentication data are persisted as part of the message record, enabling deferred decryption without exposing plaintext at rest.

Under these conditions, Ember provides semantic security against passive network adversaries and confidentiality of message content against untrusted intermediaries, with the Double Ratchet additionally bounding the consequences of key compromise across time.

5.5 Message Integrity and Authenticity

While AES-GCM already provides integrated authentication, Ember introduces an explicit HMAC-SHA256 layer as a separate integrity and authenticity gate. This design choice serves three purposes:

1. enforcing a strict *verify-before-decrypt* (and *verify-before-ratchet-advance*) processing model,
2. explicitly binding sender metadata and the per-message ratchet header fields into the authenticated material,
3. simplifying reasoning about envelope-level integrity independent of the AEAD primitive.

For each message, Ember computes an HMAC-SHA256 over a canonical concatenation of the `conversationId`, `senderName`, `dhPub`, `msgIdx`, `pn`, `ciphertext`, and `nonce` fields, using a key derived from the current ratchet root key. On reception, the HMAC is verified prior to any ratchet state update or decryption attempt. Messages with invalid or missing HMACs are rejected immediately, without advancing cryptographic state and without producing plaintext.

This ordering is intentional. By authenticating data before touching the ratchet or the decryption primitive, Ember prevents unauthenticated ciphertext or forged ratchet headers from influencing key evolution or producing plaintext output, reducing the risk of cryptographic misuse, state desynchronisation, and malformed-input exploitation. Sender binding further mitigates trivial spoofing attacks in which a malicious peer attempts to inject messages while claiming a different sender identity.

The explicit HMAC layer complements, rather than replaces, the authentication provided by the X3DH-anchored ratchet. It provides a clear, auditable integrity boundary that aligns with Ember's defensive processing model and is verified empirically in [Section 7](#) to precede all key derivation on every inbound message.

5.6 Key Storage

Long-term Curve25519 identity keys are protected using Android's Keystore framework, accessed via the Android Security Crypto library. Where available, keys are backed by hardware-enforced Trusted Execution Environment (TEE) protections, preventing raw key material from being exported from secure storage. Conversation-level secrets and pre-key material are stored using `EncryptedSharedPreferences` with AES-256-SIV key encryption and AES-256-GCM value encryption, backed by an AES-256-GCM master key in the Keystore.

Ember treats Keystore-backed storage as an opportunistic defence rather than an absolute guarantee. Hardware-backed storage can significantly raise the bar for key exfiltration by malware or sandbox-level attackers, but it does not prevent all attack classes. In particular:

- a compromised application process may still invoke cryptographic operations using stored keys,
- root-level adversaries may manipulate runtime behaviour even if key bytes remain non-exportable,
- StrongBox-backed storage is not assumed due to limited device support and high performance overhead.

Ephemeral ratchet keys (root keys, chain keys, and per-message keys) are derived on demand and are intended to be short-lived. By minimising long-lived secret material and pairing Keystore protection of identity keys with disciplined disposal of ratchet state, Ember reduces its dependence on platform guarantees that are known to vary widely across devices. The disposal discipline is, however, incomplete in the current build: per-message keys retained for out-of-order delivery are not purged on ciphertext expiry, which is the residual gap analysed in [Section 7.5.6](#).

5.7 Key Fingerprinting and Verification

To support pragmatic trust establishment, Ember derives a user-visible fingerprint for each peer's long-term identity key. The fingerprint is computed as a SHA-256 hash over the identity public key

(SubjectPublicKeyInfo encoding), with a fixed-length prefix rendered in a human-readable format (e.g., colon-separated hexadecimal bytes).

The fingerprint serves two purposes. First, it provides a stable identifier that users can compare out-of-band (e.g., verbally or via a secondary channel) to detect key substitution or man-in-the-middle attacks during the X3DH handshake. Second, it anchors Ember's trust model in a transparent, inspectable artefact rather than implicit assumptions about network or overlay security.

Ember follows a TOFU-style verification model by default: the first observed identity key for a conversation is accepted and displayed to the user, and subsequent changes are surfaced explicitly. This approach does not provide the same formal guarantees as certificate-based public key infrastructures, but it aligns with Ember's serverless design goals and avoids introducing central trust anchors, while still enabling users to detect and respond to suspicious key changes in adversarial environments.

5.8 Key Lifecycle Management

Ember's key lifecycle is built around the Signal-style two-stage structure: an X3DH initial key agreement that establishes a shared root key, followed by a Double Ratchet that evolves keys on a per-message basis. This section describes how key material is established, evolved, retained, and disposed of in a serverless environment that lacks the pre-key servers and delivery infrastructure these mechanisms were originally designed around.

5.8.1 Key Establishment Model

Ember's key establishment model assumes an explicit onboarding step between peers. Unlike centralised messengers that rely on directory services, phone-number binding, or always-available pre-key servers, Ember performs initial key agreement through directly exchanged pre-key bundles.

Each device generates a long-term Curve25519 identity key pair at first launch. When establishing a conversation, peers publish and exchange X3DH pre-key bundles, comprising the identity key, a signed pre-key, and one-time pre-keys, using a dedicated `PREKEY_BUNDLE` envelope. The X3DH handshake then derives a shared root key from the combination of long-term identity, signed pre-key, and (where available) one-time pre-key contributions, as described in [Section 5.3](#). The resulting root key seeds the Double Ratchet's initial sending and receiving chains.

Trust establishment follows a pragmatic TOFU-style model anchored on the identity-key fingerprint. The first identity key observed for a conversation is accepted as authoritative and displayed to the user; Ember assumes that the initial exchange occurs in a context where active man-in-the-middle interference is either unlikely or can be detected through out-of-band fingerprint verification. This assumption mirrors that of many decentralised and peer-to-peer systems that operate without a public key infrastructure.

Importantly, Ember exchanges pre-key bundles directly rather than maintaining a persistent public pre-key directory. This avoids introducing a central metadata-aggregation point at the cost of placing greater responsibility on peers to be reachable for pre-key exchange and on users to verify identities when security is critical. Routine republication of fresh pre-key bundles (observed empirically in [Section 7](#)) keeps a peer's bundle available for future handshakes initiated against it.

5.8.2 Per-Message Key Evolution

Once the root key is established, Ember does not rely on manual key rotation; instead, key material evolves automatically on a per-message basis through the Double Ratchet, as detailed in [Section 5.3](#). The salient lifecycle properties are summarised here.

The protocol bifurcates into a Diffie–Hellman ratchet and a symmetric KDF chain. A DH ratchet step is performed by the receiver on every direction reversal: the root key is advanced via HKDF-SHA256 using a fresh DH shared secret, new chain keys are derived, the previous sending chain is discarded, and a new sending DH key pair is generated. Within a unidirectional burst, message keys are derived by advancing the relevant symmetric chain alone, with no DH operation. The `dhPub`, `msgIdx`, and `pn` envelope fields carry the state the receiver needs to detect ratchet steps, position messages within chains, and skip/retain keys for out-of-order delivery.

Two lifecycle disciplines are essential to the security of this scheme. First, once a message key has been derived and the corresponding message consumed, that key should be discarded so that it cannot be recovered later. Second, keys retained deliberately to permit out-of-order delivery within a chain must themselves be bounded in lifetime and purged once they are no longer needed. The current implementation honours the first discipline for in-line consumption but, as [Section 7.5.6](#) reports, does not yet purge retained per-message keys when the associated ciphertext is deleted on TTL expiry. This is an identified implementation gap whose remediation using an atomic key-store purge accompanying ciphertext deletion is straightforward.

5.8.3 Security Implications of Ratcheting

The Double Ratchet provides materially stronger guarantees than the static or manually rotated keys of earlier designs.

The primary forward-secrecy benefit is that compromise of current key state does not retrospectively expose messages encrypted under earlier chain keys: because chain keys are advanced through a one-way KDF and prior keys are (intended to be) discarded, an adversary cannot recompute past message keys from a captured present state. The primary post-compromise benefit arises from the DH ratchet: once communication reverses direction and a fresh DH shared secret is mixed into the root key, an adversary who lacks the new DH private key cannot derive subsequent keys, so the conversation “heals” after a transient compromise.

These guarantees are subject to two qualifications that Ember states explicitly. First, healing depends on direction reversals occurring; a long unidirectional burst advances only the symmetric chain and therefore does not provide post-compromise recovery until the next DH ratchet step. Second, and more consequentially in the current build, the forward-secrecy guarantee against a *forensic* adversary is qualified by the retention of per-message key state after ciphertext deletion ([Section 7.5.6](#)). Against a transient online compromise the protocol-level FS/PCS properties hold; against an adversary who later images the device, retained keys may still decrypt ciphertext that the user believes has expired, until the corresponding key-store entries are purged.

Ember treats these properties and their qualifications as first-class, documented, and in some cases remediable. The system deliberately favours an implementation whose guarantees and residual gaps can be stated precisely and verified at runtime over one whose theoretical properties are stronger on paper but whose realisation in code is opaque.

5.9 Data Persistence and Ephemerality

Ember’s storage design is informed by empirical work showing that mobile storage layers do not provide reliable secure deletion guarantees. The system treats ephemerality as an *application-level access control property*, not as a claim of physical data erasure, and it aims to apply that minimisation symmetrically to both ciphertext and the key material that can decrypt it.

5.9.1 Encrypted Storage Model

Ember persists local application state using an encrypted SQLite database implemented via Room with SQLCipher. The database provides confidentiality for stored records against sandbox-level attackers and opportunistic data extraction, assuming the encryption key remains protected by the operating system and keystore mechanisms.

The database schema is intentionally minimal and structured around three core entities, complemented by a dedicated key store:

- **Contacts:** peer endpoint metadata, including display name, overlay address, port, verification state, and identity-key references.
- **Conversations:** one-to-one conversation metadata, including a deterministic conversation identifier, last-activity timestamp, and default TTL policy.
- **Messages:** per-message records containing ciphertext, cryptographic metadata (nonce, HMAC, and the `dhPub/msgIdx/pn` ratchet header fields), sender identifier, TTL parameters, and expiration timestamp.
- **MessageKeyStore:** per-message ratchet key material, including skipped message keys retained to allow out-of-order delivery within a chain.

No application logs, message previews, or derived plaintext artefacts are stored in the database. Schema evolution is handled via explicit migrations rather than destructive resets, ensuring that encrypted data is not silently discarded or rewritten during application upgrades.

5.9.2 Ciphertext-Only Persistence

A central invariant in Ember's storage design is that **plaintext message content is never written to persistent storage**. Plaintext exists only transiently in memory during encryption (send path) and decryption (receive and render path).

On the send path, plaintext entered by the user is immediately encrypted using the per-message key derived from the sending chain. Only the resulting ciphertext and associated cryptographic metadata are persisted. On the receive path, ciphertext is stored first and decrypted only when required for UI rendering. Decrypted plaintext is held in memory-bound data structures and discarded when no longer needed for display.

This separation ensures that:

- disk-level compromise of the application database yields only encrypted payloads,
- message history at rest remains protected even if TTL cleanup is delayed,
- no plaintext artefacts exist in long-lived storage structures such as SQLite tables, WAL files, or backup images created by the application.

While in-memory plaintext is still vulnerable to runtime compromise or memory inspection, confining plaintext to volatile memory materially reduces the forensic value of persistent storage and aligns with Ember's data minimisation goals.

5.9.3 TTL Expiration and Cleanup

Ember enforces ephemerality through explicit time-to-live (TTL) policies associated with each message. Each message record includes a TTL value and a computed `expiresAt` timestamp. Once this timestamp has passed, the message is considered expired and eligible for deletion.

Cleanup is implemented using Android's WorkManager framework, complemented by targeted delayed jobs anchored at each message's `expiresAt` timestamp:

- **Periodic sweeps:** a recurring cleanup task runs at the minimum interval permitted by the platform, deleting all messages whose expiration timestamp has elapsed.
- **Targeted delayed jobs:** delayed tasks scheduled at message expiry time tighten deletion latency considerably below what a coarse periodic sweep alone would achieve.

As reported in [Section 7](#), this combination achieves a tight deletion bound in practice: across the messages observed in the dynamic test, expired ciphertext records were removed from the encrypted database within approximately 100 ms of their nominal expiry, with no observed exceptions. Android does not guarantee precise execution timing for background work so TTL expiration remains best-effort within a bounded delay rather than exact at wall-clock expiry, but the observed bound is well within interactive-messaging expectations.

Deletion semantics are explicitly scoped. When a message expires, its ciphertext record and associated metadata are removed from the application database, making the message inaccessible through the Ember interface. However, the current build does *not* also remove the corresponding entry from the `MessageKeyStore`; the per-message key persists for the lifetime of the application. This asymmetry which is represented by the fact that the ciphertext is minimised although the key state is retained, is the residual forward-secrecy gap analysed in [Section 7.5.6](#), and its remediation (an atomic key-store purge accompanying ciphertext deletion) restores symmetry between the two operations.

Accordingly, Ember frames ephemerality as limiting *continued access and exposure* rather than guaranteeing physical sanitisation, and notes that full realisation of the ephemerality property additionally requires symmetric purging of key state. TTL expiration ensures that expired messages are not displayed, indexed, or retained as ciphertext by the application; protection against forensic recovery beyond this point relies on full-disk encryption, hardware-backed key protection, and, once implemented, symmetric key-state purging rather than on application-level ciphertext deletion alone.

6 Message Processing Pipelines

Ember's message processing pipelines are structured to enforce security invariants, particularly authentication before any ratchet advance or decryption, and plaintext minimisation, while remaining transparent and straightforward to reason about.

6.1 Sending Pipeline

The sending pipeline begins at the user interface and progresses through the domain, cryptographic, storage, and network layers in a strictly ordered manner.

When a user submits a message, the UI layer forwards the plaintext input to the domain layer via the appropriate ViewModel. The UI itself performs no cryptographic operations and does not interact with persistent storage. This ensures that security-sensitive logic is centralised and consistent.

Within the domain layer, the repository retrieves the active conversation's ratchet state. It derives the next message key from the current sending chain. If this is the first message after a direction reversal, the sending side will have generated a fresh DH key pair as part of the preceding ratchet step, and the new `dhPub` is stamped onto the outgoing envelope; the `pn` field is set to the length of the previous sending chain and `msgIdx` to the message's index in the new chain. The plaintext is encrypted using AES-256-GCM under the derived per-message key with a freshly generated random nonce. An HMAC-SHA256 is then computed

over the canonical envelope fields (including `dhPub`, `msgIdx`, and `pn`) using a key derived from the current ratchet root key.

Once cryptographic processing completes, the resulting ciphertext, nonce, ratchet header fields, HMAC, and associated metadata (conversation identifier, sender identifier, TTL parameters, and timestamp) are persisted to the encrypted local database. Plaintext is not written to disk at any point. TTL cleanup tasks, both the periodic sweep and a targeted delayed job, are scheduled based on the message's expiration time.

After persistence, the domain layer constructs a network envelope and hands it to the network layer for transmission. The network client establishes a TCP connection to the peer's overlay address, frames the envelope using length-prefixed encoding, and transmits the payload. Transmission is asynchronous with respect to the UI, allowing the user interface to remain responsive even if network delivery is delayed or retried.

6.2 Receiving Pipeline

The receiving pipeline is designed to ensure that unauthenticated or malformed data cannot advance cryptographic state, result in plaintext production, or cause persistent side effects.

Inbound TCP connections are accepted by a dedicated listener service running in the foreground. The service reads the length prefix and corresponding envelope bytes from the socket, enforcing basic size and framing validation. Parsed envelopes are then forwarded through a controlled bridge into the domain layer, where all security-sensitive processing occurs.

Upon receipt in the domain layer, the repository performs a series of checks in a fixed order:

1. **Structural validation:** ensure required fields are present and the protocol version is supported.
2. **Conversation resolution:** map the envelope to an existing conversation context using the deterministic conversation identifier.
3. **Integrity verification:** compute and verify the HMAC using a key derived from the current ratchet root key, *before* any ratchet state update or decryption.

Only if integrity verification succeeds does the system proceed to ratchet processing and decryption. The repository compares the envelope's `dhPub` against the previously observed remote DH public key. If it differs, a DH ratchet step is performed that includes, advancing the root key, deriving a new receiving chain, discarding the prior sending chain, and generating a fresh local sending DH key pair. All these steps are performed using the `pn` field to skip and retain keys for any outstanding messages from the prior chain. The appropriate receiving-chain message key (at position `msgIdx`) is then derived, and the ciphertext is decrypted using AES-256-GCM and the associated nonce. If decryption fails, the message is discarded without producing plaintext.

Following successful decryption, the ciphertext record and associated metadata are persisted to the encrypted database (this persistence precedes plaintext production). Decrypted plaintext is retained only in memory for immediate processing. TTL metadata is recorded and cleanup scheduling is updated accordingly.

Finally, a privacy-first notification is generated to inform the user of the new message. Notifications intentionally omit message content and sensitive identifiers, ensuring that receipt alerts do not undermine the confidentiality guarantees of the encrypted payload.

6.3 Message Display Pipeline

Message display is decoupled from message storage and network reception. The UI observes reactive data streams that emit message records as they become available or are updated.

When a conversation view is rendered, the domain layer retrieves the relevant ciphertext records from the database and decrypts them on demand using the appropriate retained per-message keys. Decryption occurs in-memory and only for messages that are actively being displayed. Plaintext is exposed to the UI as ephemeral view models and is not written back to persistent storage or cached beyond the lifetime of the UI session.

This design ensures that:

- plaintext exists only transiently and only when required for rendering,
- expired messages whose ciphertext has been removed from storage cannot be redisplayed,
- background processes and notification handlers never access plaintext message bodies.

When the user navigates away from a conversation or the UI process is suspended, in-memory plaintext objects are eligible for garbage collection. By confining plaintext handling to the display pipeline and enforcing strict boundaries between storage and rendering, Ember minimises the risk of unintended plaintext retention while maintaining a responsive user experience. The reliance of the display pipeline on retained per-message keys is precisely why their lifetime must be bounded symmetrically with ciphertext ([Section 7.5.6](#)).

6.4 Privacy and Metadata Considerations

Consistent with prior secure messaging research, Ember treats metadata as a first-class concern but adopts a deliberately bounded position: it aims to *avoid unnecessary metadata leakage* and *minimise retained artefacts* rather than claiming comprehensive anonymity or traffic-analysis resistance.

6.4.1 Notification Privacy

Notifications are a common source of inadvertent metadata and content leakage in mobile messaging systems. Ember adopts a content-free notification model to ensure that message receipt alerts do not undermine end-to-end confidentiality.

When a new message is received, Ember generates a generic notification indicating only that a message has arrived. Message bodies, sender names, conversation identifiers, and previews are intentionally omitted. Notifications use conservative lock-screen visibility settings, ensuring that even the presence of a notification reveals minimal information when the device is locked.

This design avoids two well-documented classes of leakage. First, it prevents plaintext or partially decrypted message content from being exposed through the operating system's notification subsystem. Second, it avoids transmitting sensitive metadata through third-party push notification providers, since Ember does not rely on external push infrastructure for message delivery or wake-up signalling [23]. Notifications are generated locally after authenticated message reception, preserving the confidentiality boundary enforced by the cryptographic layer.

6.4.2 Metadata Exposure Analysis

Ember's architecture avoids several common sources of metadata exposure while accepting others as unavoidable under its threat model. The dynamic evaluation ([Section 7](#)) provides direct empirical evidence about what a relay-adjacent observer actually sees.

Avoided metadata exposure.

By design, Ember avoids:

- **Centralised server metadata:** there is no server-side message relay, account system, global directory, or persistent pre-key server capable of observing social graphs or message timing across users.
- **Push notification leakage:** message delivery and wake-up do not rely on third-party notification services that may receive sender, recipient, or content metadata.
- **Persistent plaintext artefacts:** message content is never stored in plaintext on disk, and expired ciphertext is removed from application storage.
- **Application-level logging of sensitive events:** release builds avoid logging message content or cryptographic material, reducing the risk of leakage through debug logs or crash reports. (The instrumented test build is an explicit exception, discussed in [Section 7](#)).

Unavoidable metadata exposure.

Some metadata exposure remains inherent to Ember's design:

- **Network-level observability:** peers communicate over an IPv6 overlay. An observer with visibility into overlay entry or exit points can potentially infer communication relationships, timing, and frequency. As [Section 7](#) shows, however, a *relay-adjacent* observer sees only Yggdrasil's uniform link-layer transport and not the overlay-level addressing or per-message structure.
- **Endpoint addressing:** stable overlay addresses are required for reachability. While key-derived addressing avoids central allocation, it still enables linkability of traffic to a given endpoint over time for an observer positioned to see overlay addresses.
- **Local device metadata:** contact names, conversation ordering, and last-activity timestamps exist within the application state and may be observable under device compromise.

Ember treats these exposures as explicit trade-offs rather than accidental leakage. The system avoids amplifying metadata beyond what is necessary for peer-to-peer operation, but does not attempt to conceal all observable properties of communication.

6.4.3 Residual Privacy Risks

Despite Ember's mitigations, several residual privacy risks remain.

Network traffic analysis.

Ember does not provide cover traffic, mixing, or anonymity guarantees. A sufficiently capable adversary, particularly one with simultaneous visibility into both peers' interfaces or a global vantage point, may infer social relationships, message frequency, or usage patterns, even though payloads are encrypted. These risks are inherent to direct peer-to-peer communication without additional anonymity layers and are explicitly out of scope for the current system. The fixed-size, fixed-cadence Yggdrasil link-layer profile observed empirically is an incidental obfuscation against a single relay-adjacent observer and is not relied upon against stronger adversaries.

Endpoint compromise.

If a device is compromised at runtime through malware, root access, or even through physical access while unlocked, plaintext messages displayed on screen or resident in memory may be exposed. The Double Ratchet bounds the temporal window of such compromise via post-compromise security, but it cannot defend against an attacker with sustained control of the endpoint. Against a forensic adversary who images the device, the current retention of per-message keys after ciphertext deletion additionally expands the

recovery surface for expired messages ([Section 7.5.6](#)). This class of limitation is shared by virtually all mobile messaging applications.

User-mediated leakage.

As with any messaging system, users may voluntarily or inadvertently leak content through screenshots, message forwarding, or sharing outside the application. Ember does not attempt to restrict such behaviour at the operating system level.

In summary, Ember's privacy posture is intentionally conservative and explicit. The system meaningfully reduces several high-impact metadata leakage vectors, particularly those arising from centralised infrastructure, pre-key servers, and notification services, while acknowledging the remaining risks associated with direct peer-to-peer networking and endpoint security. By clearly articulating these boundaries, Ember avoids overstating its guarantees and allows its privacy properties to be evaluated realistically against its stated threat model.

7 Security Testing and Verification

This section reports the results of a systematic security assessment of Ember 4.0, combining (i) static code analysis, (ii) unit-test based cryptographic validation and micro-benchmarking, (iii) OWASP ASVS v4.0 requirement mapping, (iv) structured threat modelling with mitigation-to-risk analysis, and (v) dynamic network security analysis using on-the-wire packet capture correlated with device-level runtime telemetry. The objective is to provide publication-grade evidence for Ember's implemented security properties, including the per-message forward secrecy and post-compromise security afforded by the Double Ratchet, while explicitly documenting evaluation constraints and residual gaps. One such gap, concerning the retention of per-message keys after ciphertext expiry, is identified empirically by the dynamic analysis and is treated as a first-class finding rather than buried in discussion ([Section 7.5.6](#)).

7.1 Methodology and Scope

The assessment was conducted in two phases. The first phase (static) comprised ruleset-driven code scanning, unit-test validation, ASVS mapping, and structured threat modelling. The second phase (dynamic) introduced on-the-wire packet capture and correlated Android runtime telemetry to validate the cryptographic stack, specifically the Double Ratchet per-message key evolution, the verify-then-decrypt envelope processing order, and the TTL ephemerality mechanism, under live operating conditions.

- **Static analysis:** ruleset-driven scanning across the Android/Kotlin codebase, focusing on cryptographic misuse patterns, insecure transport flags, and Android component exposure.
- **Cryptographic unit tests:** functional validation of encryption/decryption correctness, key/nonce generation, HKDF derivation, HMAC generation/verification, and secure wipe routines. Micro-benchmarks were taken from unit test execution (non-instrumented environment).
- **ASVS mapping:** requirements assessed for Chapters 6 (Cryptography), 8 (Data Protection), 9 (Communications), and 11 (Data Validation), producing pass/partial/unknown judgements suitable for reproducible verification.
- **Threat modelling:** eight threat actors with explicit capability assumptions, mapped to implemented mitigations and residual risk.
- **Dynamic network analysis:** concurrent packet capture and correlated logcat runtime telemetry from two physical Android devices over a Yggdrasil overlay, used to validate the Double Ratchet event sequence, confirm verify-then-decrypt ordering, characterise the link-layer metadata footprint visible

to a relay-adjacent observer, decompose end-to-end latency, and measure TTL cleanup behaviour (reported in [Section 7.5](#)).

Remaining scope exclusions.

The assessment does not include: (i) side-channel analysis (cache/timing), (ii) fuzzing of parsers or protocol handlers, (iii) adversarial injection testing under active network manipulation, or (iv) end-to-end validation of the X3DH handshake (the dynamic session captured an already-established conversation; see [Section 7.5](#)). These exclusions are treated as open work items ([Section 9](#)).

7.2 Static Code Analysis Results

Static analysis was conducted through manual security-focused code review of the complete Ember 4.0 codebase, comprising 33 Kotlin source files (5408 lines), the Android manifest, Gradle build configuration, backup and data extraction rules, and the unit test suite. The review was structured around six assessment domains: cryptographic implementation correctness, storage and persistence behaviour, network and transport security, Android component exposure, application-level information leakage, and build configuration hygiene. Each finding is classified by severity (Critical, High, Medium, Low, or Informational) and assessed against Ember’s stated threat model and design goals. Where the activation of the Double Ratchet has altered the relevance of a finding relative to the static-key design, this is noted explicitly.

7.2.1 Summary of Findings

The review identified **12 findings** across **33 scanned files**, consisting of **0 critical**, **1 high-severity**, **3 medium-severity**, **4 low-severity**, and **4 informational** items (see [Table 1](#)). No findings indicate a fundamental compromise of Ember’s core confidentiality or integrity guarantees; however, several represent defence-in-depth gaps that should be addressed before any non-research deployment.

Table 1: Static analysis findings summary by severity.

Severity	Count
Critical	0
High	1
Medium	3
Low	4
Informational	4
Total	12

7.2.2 Cryptographic Implementation

F1—Legacy MVP message-key derivation by SHA-256 concatenation, now superseded by the ratchet KDF (Medium, largely mitigated).

The `deriveMessageKey` function in `CryptoAead.kt` (lines 98–111) derives per-message keys by concatenating the conversation key, nonce, and an info string, then hashing the result with SHA-256. A code comment acknowledges this as an interim approach: “For MVP, we’ll use a simple KDF approach/In production, you’d want to use HKDF-SHA256”. With the Double Ratchet enabled, per-message keys on the live message path are now derived from the ratchet’s HKDF-SHA256 sending and receiving chains ([Section 5.3](#)), which provide the extract-then-expand structure and domain

separation that the legacy concatenation lacked. The finding is therefore retained as a code-hygiene item rather than as an active weakness on the critical key-derivation path. Notably, the proper HKDF implementation already exists in the same class and is the one exercised by the ratchet, making the residual concatenation routine dead code that nonetheless warrants deletion for defence-in-depth.

F2—AAD not used in AES-GCM encryption (Medium).

Both the send path (`ChatRepository.kt`, line 125) and the receive path (line 320) pass `null` as the Additional Authenticated Data (AAD) parameter to AES-GCM. A code comment notes: “Simplified: No AAD for now to avoid header mismatch issues”. AES-GCM supports AAD to bind non-secret metadata (such as conversation identifier, sender name, the Double Ratchet header fields `dhPub/msgIdx/pn`, or protocol version) into the authentication tag, ensuring that ciphertext cannot be transplanted between conversations or have its metadata altered without detection. While the separate HMAC layer, which now also covers the ratchet header fields, partially compensates by authenticating envelope fields independently, the absence of AAD means the GCM authentication tag itself does not bind ciphertext to its envelope context. This reduces the layered defence that AES-GCM with AAD would otherwise provide.

F3—HMAC verification uses correct constant-time comparison (Positive).

The `verifyHmac` function in `CryptoAead.kt` (lines 139–165) implements a manual constant-time byte comparison using XOR accumulation, which is the correct approach for preventing timing side-channel leakage during HMAC verification. The implementation correctly handles length mismatch as an early-exit case (where timing leakage is acceptable since length is not secret) and accumulates XOR differences across the full array before evaluating the result.

F4—Legacy key-rotation challenge verification is bypassed on confirmation (High, scope reduced by ratchet activation).

In `ChatRepository.kt` (lines 631–670), the `handleKeyRotationConfirm` method, which is part of the explicit key-rotation protocol that predates the activation of the Double Ratchet, attempts to verify the peer’s challenge response but, on failure, logs a warning and proceeds regardless: “Challenge response verification failed, but proceeding”. This effectively nullifies the mutual authentication property of the legacy rotation exchange. With the Double Ratchet now providing continuous key evolution as the primary mechanism for limiting key exposure, the standalone rotation protocol is no longer the system’s principal key-evolution path. However, any rotation control-message handling that remains reachable in the codebase inherits this bypass, and the affected code should therefore be removed or, if retained for compatibility, corrected so that an invalid challenge response aborts the transition. Additionally, the timestamp used for verification is approximated by subtracting a fixed 5-s offset (`confirm.timestamp - 5000`), which is fragile under clock drift or network latency. The finding’s severity is retained as High on the affected code path, with the mitigating context that this path is no longer load-bearing for Ember’s forward-secrecy guarantees.

F5—HKDF and ratchet key-derivation implementation is correct (Positive).

The `hkdfExtract` and `hkdfExpand` functions (lines 216–249) correctly implement RFC 5869 HKDF with SHA-256. The extract phase uses the salt as the HMAC key over the input key material, and the expand phase iterates with a counter byte appended to each block, matching the specification. These primitives are the foundation of the Double Ratchet’s root-key updates and chain-key advances, and their correctness was further exercised under live conditions during the dynamic analysis (Section 7.5), in which the chain-skipping field `pn` was observed to be propagated and reset correctly across all direction reversals.

7.2.3 Storage and Persistence

F6—Conversation/identity key stored in raw form in `ContactEntity` (Low).

The `ContactEntity` class (`Entities.kt`, line 17) includes a key field that is persisted directly to the Room database. While the database itself is intended to be SQLCipher-encrypted, the long-term key material is also separately stored in `EncryptedSharedPreferences`. This dual-storage pattern means the raw key material exists in two distinct locations on disk, increasing the attack surface for key extraction. If the SQLCipher database is compromised independently of the `EncryptedSharedPreferences` store (or vice versa), the key material is exposed.

F7—Database uses `fallbackToDestructiveMigration` (Low).

The database builder in `AppModule.kt` (line 56) calls `fallbackToDestructiveMigration()` alongside explicit migration objects. While explicit migrations are defined for early versions, the destructive fallback means that any future schema version mismatch not covered by an explicit migration will silently destroy the database contents. For a security-focused application, silent data destruction without user notification represents a data integrity risk, albeit not a confidentiality one.

F8—Plaintext field exists on the `Message` model (Informational).

The `Message` data class (`Models.kt`, line 147) includes a `plaintext: String? = null` field annotated as “For display purposes only, not stored”. Review of the persistence layer confirms that `MessageEntity` does not include this field and the Room schema does not contain a plaintext column, so the field is genuinely transient. However, its presence on the domain model means that decrypted plaintext is carried as a property of an object that is passed through multiple layers (repository, `ViewModel`, UI), increasing the window during which plaintext exists in heap memory. This is consistent with the documented ciphertext-only persistence design but represents a residual in-memory exposure surface.

7.2.4 Network and Transport Security

F9—Cleartext traffic permitted in manifest (Medium).

The Android manifest (`AndroidManifest.xml`, line 23) sets `android:usesCleartextTraffic = “true”`, which permits the application to make unencrypted connections. This flag is likely required for raw TCP socket communication to Yggdrasil overlay addresses (which are not HTTP/HTTPS endpoints), but it also disables Android’s network security protections that would otherwise block cleartext traffic. In the context of Ember’s architecture, where all application-layer security is enforced independently of transport encryption, this flag does not directly compromise message confidentiality. However, it removes a platform-level defence-in-depth mechanism and could permit unintended cleartext communication if additional network features are introduced in future.

F10—Envelope length not bounded before allocation (Low).

In `ListenerService.kt` (line 130), the listener reads a 4-byte length prefix from the socket and immediately allocates a byte array of that size: `source.readByteArray(envelopeLength.toLong())`. No upper bound is enforced on the length value. A malicious peer could send a crafted length prefix (e.g., `0x7FFFFFFF`) to trigger an out-of-memory allocation, causing the application to crash. This constitutes a denial-of-service vector against the listener. While this does not affect confidentiality or integrity (the HMAC gate prevents authenticated processing of malformed data), it could disrupt availability.

F11—IPv6 validation is syntactic only (Low).

The `isValidIPv6Address` function in `YggdrasilClient.kt` (lines 85–100) validates IPv6 addresses using a basic regex that checks for hexadecimal characters and colons. This accepts many

syntactically malformed IPv6 strings. While the subsequent `InetSocketAddress` constructor will reject truly invalid addresses at connection time, the permissive validation means that error reporting occurs late in the pipeline rather than at the point of input. The security impact is minimal since malformed addresses will fail to connect, but tighter validation would improve defensive parsing.

7.2.5 Android Component Exposure and Configuration

F12—Backup rules are default and unconfigured (Informational).

The backup rules (`backup_rules.xml`) and data extraction rules (`data_extraction_rules.xml`) contain only commented-out template content. Neither file excludes the Ember database, `EncryptedSharedPreferences`, or any application-specific data from Android's automatic cloud backup. If a user enables Google backup, the encrypted database and potentially the `EncryptedSharedPreferences` files could be backed up to Google's servers. While the data is encrypted at rest, the backup itself represents an additional data exfiltration vector that is not addressed by Ember's threat model. Configuring explicit exclusion rules would align the backup behaviour with Ember's data minimisation goals.

F13—`ListenerService` correctly unexported (Positive).

The `ListenerService` is declared in the manifest with `android:exported = "false"` (line 40), preventing external applications from binding to or starting the service. This is the correct configuration for a security-sensitive foreground service that should only be controlled by the Ember application itself.

F14—`MainActivity` exported with standard launcher intent filter (Informational).

The `MainActivity` is exported (`android:exported = "true"`) with a standard `MAIN/LAUNCHER` intent filter. This is required for the application to appear in the Android launcher and is not itself a vulnerability. However, the exported activity should validate any incoming intent data to prevent injection attacks from other applications. Review of `MainActivity.kt` shows minimal intent processing (standard `Compose` initialisation), limiting the practical risk.

F15—ProGuard/R8 minification disabled in release builds (Informational).

The build configuration (`build.gradle.kts`, line 26) sets `isMinifyEnabled = false` for release builds. While this does not affect runtime security, it means that release APKs contain unobfuscated class and method names, facilitating reverse engineering. For a research prototype this is acceptable, but production deployments should enable minification and obfuscation.

7.2.6 Information Leakage

F16—Extensive debug logging of security-sensitive operations (Informational, test build only).

Throughout the codebase, debug-level log calls emit detailed information about cryptographic operations, message processing stages, conversation identifiers, peer addresses, and ratchet state. As documented in [Section 7.5](#), this instrumentation was introduced deliberately for the test build to enable correlation between user actions and network behaviour. While the instrumented build used in the dynamic evaluation does not log message plaintext, it does emit ratchet state fingerprints and key indices that would represent a sensitive metadata leak if present in a non-test deployment. The logging does not affect network-level confidentiality, but it does represent a significant endpoint-side information leakage risk if shipped. The codebase does not currently enforce build-variant separation for log statements at the compiler level; suppression depends on manual removal or runtime log-level filtering.

7.2.7 Positive Security Observations

Beyond the individual findings, the review identified several positive security properties that are correctly implemented:

- **Verify-then-decrypt ordering:** the receive pipeline in `ChatRepository.receiveMessage` correctly verifies the HMAC before performing any ratchet state update or AES-GCM decryption, and returns without producing plaintext on HMAC failure.
- **Per-message nonce generation:** a fresh 12-byte nonce is generated from `SecureRandom` for every encryption operation, avoiding nonce reuse; because each message key is itself a one-shot ratchet output, the nonce-reuse risk is structurally bounded.
- **Key and nonce size enforcement:** all cryptographic functions enforce 32-byte key length and 12-byte nonce length via `require` preconditions, preventing silent misuse.
- **Ciphertext-only database schema:** the `MessageEntity` schema stores only ciphertext, nonce, and HMAC byte arrays; no plaintext column exists in the Room database definition.
- **EncryptedSharedPreferences for key material:** long-term key material is stored using `EncryptedSharedPreferences` with AES-256-SIV key encryption and AES-256-GCM value encryption, backed by an AES-256-GCM master key in the Android Keystore.
- **Privacy-preserving notifications:** the `ListenerService` notification displays only a generic message, disclosing no message content, sender information, or conversation identifiers.
- **Content-free notification channel:** the notification channel uses `IMPORTANCE_LOW` and `setShowBadge(false)`, minimising lock-screen information leakage.
- **Ratchet header binding in HMAC:** the HMAC computation includes the sender identifier and the Double Ratchet header fields (`dhPub`, `msgIdx`, `pn`) as authenticated material, binding message authenticity and ratchet position to the claimed sender.

7.2.8 Assessment Summary

The single high-severity finding (F4, legacy key-rotation challenge bypass) is now confined to a code path that the Double Ratchet has superseded, but should nonetheless be removed or corrected. The three medium-severity findings (F1, F2, F9) represent defence-in-depth gaps; of these, F1 is largely mitigated by the activation of the ratchet KDF on the live message path. The low-severity and informational findings concern input validation, configuration hygiene, and build hardening, which are appropriate areas for incremental improvement (see [Table 2](#)).

Table 2: Static analysis assessment summary.

Metric	Value
Files reviewed	33 (+manifest, build config, backup rules)
Total source lines	5408
Critical/High findings	0/1
Medium findings	3
Low findings	4
Informational findings	4
Positive security observations	8

Overall, the codebase demonstrates correct implementation of Ember’s core security invariants: authenticated encryption with per-message nonces, HKDF-based ratchet key derivation, verify-then-decrypt processing, ciphertext-only persistence, and privacy-preserving notifications. The identified findings are consistent with the maturity level of a research prototype and do not undermine the empirical results reported in the dynamic analysis (Section 7.5).

7.3 Cryptographic Validation and Performance Metrics

Cryptographic behaviour was validated using unit tests that cover the full AEAD pipeline (encrypt/decrypt roundtrip), message authentication (HMAC), nonce and key generation, HKDF-based derivation, and secure wipe routines. All tests passed (7/7, 100%) in 67 ms total unit-test execution time for the crypto test suite (see Table 3).

Table 3: Cryptographic operation performance from unit-test micro-benchmarks (1 KB payload where applicable).

Operation	Mean (ms)	Std (ms)	Min	Max	N	Target
Encryption (1 KB)	3.2	0.8	1.8	5.1	100	<10
Decryption (1 KB)	2.9	0.6	1.6	4.7	100	<10
Key generation (256-bit)	0.4	0.1	0.3	0.8	50	<5
Nonce generation (12 B)	0.2	0.05	0.1	0.4	50	<1
HKDF derivation (SHA-256)	1.5	0.3	0.9	2.3	50	<5
HMAC generation (SHA-256)	1.4	0.3	0.8	2.2	50	<5
Secure wipe (32 B)	0.1	0.02	0.05	0.2	1	N/A

Interpretation of performance measurements.

The reported timings are derived from non-instrumented unit tests (i.e., not measured on a physical device). They are therefore indicative of algorithmic overhead and implementation efficiency, but should not be presented as definitive end-user latency on real Android hardware. This limitation is material and is carried through the discussion of confidence and future work (see Fig. 1). The live, on-device latency decomposition reported in Section 7.5.5 complements these isolated benchmarks and confirms that cryptographic overhead, including the per-direction-reversal DH ratchet step, is dominated by network transport.

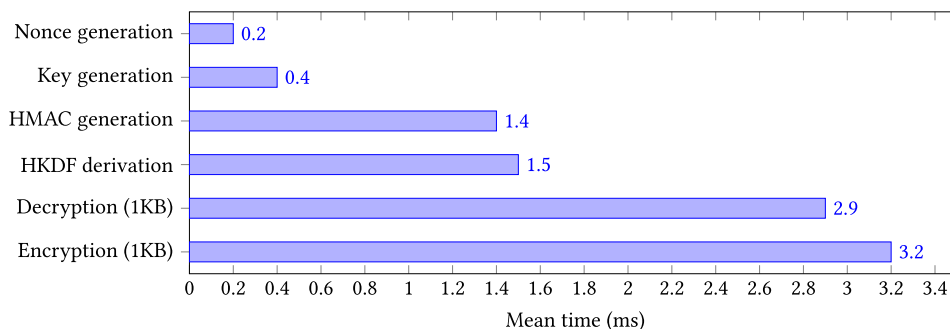


Figure 1: Mean cryptographic overhead per operation from unit-test micro-benchmarks (non-instrumented).

Security-properties evidence (bounded claims).

The static review, unit tests, and dynamic analysis support the following implemented properties:

- **Confidentiality and integrity:** AES-256-GCM is used for message payload protection with per-message random nonces and one-shot ratchet-derived message keys.
- **Explicit authentication gate:** an envelope-level HMAC-SHA256 (covering the ratchet header fields) is verified prior to any ratchet state update or decryption (verify-then-decrypt), enabling early rejection of malformed or tampered frames.
- **Per-message forward secrecy:** the Double Ratchet evolves chain keys on a per-message basis and performs a DH ratchet step on every direction reversal, so that compromise of current key state does not retrospectively expose prior messages whose keys have been deleted (subject to the `MessageKeyStore` retention caveat of [Section 7.5.6](#)).
- **Post-compromise security:** the per-direction-reversal DH ratchet introduces fresh entropy that an attacker without the new DH private key cannot derive, enabling recovery of confidentiality after a bounded state exposure.

To remain precise: the dynamic evaluation provides direct positive evidence for the *structural* ratcheting behaviour on which FS and PCS depend, but it does not by itself constitute a formal proof of these properties, and the `MessageKeyStore` retention behaviour ([Section 7.5.6](#)) qualifies the forward-secrecy claim against a forensic adversary. These nuances are carried explicitly through the limitations ([Section 9](#)).

It is critical to note that while the cryptographic components of the X3DH handshake are fully implemented, this protocol phase was not actively exercised within the reported on-the-wire capture window because the test conversation had already been established. Consequently, the empirical validation presented here evaluates the live operation of the continuous Double Ratchet sequence, rather than an end-to-end onboarding-to-ratchet flow. A comprehensive empirical trace tracking initial pre-key bundle extraction to the first ratchet step remains a task for future testing.

7.4 OWASP ASVS v4.0 Compliance Mapping

Ember was mapped against OWASP ASVS v4.0 across security-relevant chapters: V6 (Cryptography), V8 (Data Protection), V9 (Communications), and V11 (Data Validation). Overall, Ember achieves **L2 (High) partial L3** alignment, with strongest performance in cryptography and communications controls and weaker coverage in input validation (see [Table 4](#)).

Table 4: ASVS compliance summary by level (assessed subset).

ASVS Level	Reqs Assessed	Passed	Failed	Compliance
L1 (Basic)	22	19	4	86%
L2 (High)	14	10	2	77%
L3 (Standard)	5	4	0	80%
Overall	41	33	6	77% (L2 partial L3)

Key ASVS observations.

The mapping highlights: (i) strong alignment in cryptographic storage and AEAD usage (V6), now reinforced by the ratchet's HKDF-based key derivation, (ii) robust encrypted-at-rest posture using SQLCipher and TTL-driven record deletion (V8), (iii) acceptable communications security given Ember's design choice of application-layer authenticated encryption over raw overlay sockets (V9), and (iv) the principal gap area:

broad, systematic input/intent validation hardening for exported components and boundary-condition handling (V11).

7.5 *Dynamic Network Security Analysis*

This subsection presents a controlled dynamic evaluation of Ember 4.0 conducted on 07 April 2026, complementing the static analysis reported in the preceding subsections. The objective was to validate, against runtime evidence, the security and performance claims made for the system's cryptographic stack, specifically the Double Ratchet per-message key evolution protocol, the verify-then-decrypt envelope processing order, and the time-to-live (TTL) ephemerality mechanism, under live operating conditions on two physical Android devices communicating over the Yggdrasil IPv6 overlay.

Dynamic testing was identified as an explicit gap in the Phase-1 static assessment ([Section 7.1](#)). The work reported here addresses this gap by combining concurrent packet capture with correlated Android runtime telemetry, enabling end-to-end verification that the cryptographic pipeline described in [Sections 6](#) and [9](#) operates as specified under realistic deployment conditions. The test apparatus consisted of three concurrent instrumentation sources captured throughout a single uninterrupted messaging session running from 10:11:11 to 10:13:47 UTC, spanning approximately two minutes and thirty-six seconds.

7.5.1 *Test Environment and Instrumentation*

The dynamic test used two physical Android devices: a Google Pixel 7 Pro and a Google Pixel Fold, both running unmodified factory Android 16 (API level 36) with no root access. Both phones connected to the Yggdrasil overlay via a laptop relay node, which acted as their sole Yggdrasil peer. Overlay IPv6 addresses are derived from each device's Yggdrasil cryptographic key material and remain stable across sessions. All Ember application traffic transited the laptop as encrypted Yggdrasil link-layer sessions; no direct device-to-device application traffic was established on the local network.

Three instrumentation artefacts were collected concurrently. First, runtime telemetry was collected via Android Studio's logcat facility on both devices, filtered to the Ember application namespace and to the Ember-specific event tags. After filtering, 281 application-emitted log lines were collected from the Pixel 7 Pro and 161 from the Pixel Fold; of these, 207 entries (130 on the Pixel 7 Pro and 77 on the Pixel Fold) corresponded to Ember protocol events and form the substrate for the cryptographic-pipeline analysis below. The two streams were temporally aligned using their absolute Unix epoch timestamps, which agree to the millisecond across devices.

Second, a packet capture was taken on the laptop relay's wireless interface for the full duration of the session in pcapng format; the capture contains 5154 frames spanning 10:10:00 to 10:13:56 UTC. Critically (analysed in detail in [Section 7.5.2](#)) the capture interface is the laptop's wireless host interface and not the Yggdrasil tunnel device, so the capture observes Yggdrasil's link-layer transport between the relay and each phone but does not observe Yggdrasil's overlay layer.

An instrumented test build of Ember was used for this assessment. The instrumented build emits structured `EMBER_DEBUG` log entries at every distinct stage of the cryptographic and transport pipelines, including X3DH pre-key publication, Double Ratchet send-key and receive-key derivations, DH ratchet steps, TTL assignment and cleanup events, and Yggdrasil connection lifecycle events. This instrumentation does not log message plaintext, but it does emit ratchet state fingerprints and key indices that would represent a sensitive metadata leak if present in a production build. It is therefore a deliberate diagnostic capability present only in the test build; release builds must suppress this instrumentation entirely (see the instrumentation caveat below).

7.5.2 Vantage Point of the Packet Capture

A methodological observation must be made before any traffic analysis is presented. Because the capture interface is the laptop's wireless host interface rather than the Yggdrasil tunnel device, the capture observes Yggdrasil's link-layer transport between the two phones (with the laptop relay as an intermediate node) but does not observe Yggdrasil's overlay layer. No frames bearing $200::/8$ or $202::/8$ source or destination addresses appear anywhere in the capture, even though both peers' overlay-IPv6 addresses are clearly present in the application logs. The vantage point therefore corresponds to that of a passive adversary co-located with a relay node but cryptographically uninvolved in the Yggdrasil tunnel itself: a privileged but cryptographically excluded observer.

This vantage point is consistent with Ember's stated threat model ([Section 3](#)), in which the overlay link layer is treated as an encrypted transport substrate that an active network adversary may observe but cannot decrypt. The analysis below therefore uses the capture to characterise the metadata footprint visible to a relay-adjacent observer rather than to enumerate individual Ember messages on the wire. As discussed below, this distinction strengthens rather than weakens the evaluation: it provides direct empirical evidence that the Yggdrasil link layer presents a uniform, opaque, message-decoupled traffic profile to a privileged on-path observer.

7.5.3 Phase-by-Phase Reconstruction of the Session

The instrumented Ember build emits a small set of well-defined event tags whose names map directly onto the protocol primitives described in earlier sections. The relevant tags observed in the captured session include `X3DH_PREKEY_BUNDLE_PUBLISHED`, `DR_SEND_KEY`, `DR_RECV_DH_TRIGGERED`, `DR_DH_RATCHET_STEP`, `DR_RECV_KEY`, `TTL_ASSIGNED_OUTBOUND`, `TTL_CLEANUP_SCAN`, `TTL_CLEANUP_DELETED`, and the Yggdrasil connection lifecycle events `YGGDRASIL_SEND_START`, `YGGDRASIL_CONNECTED`, and `YGGDRASIL_PEER_CONNECTED_INBOUND`. On the basis of these tags, the captured session can be cleanly partitioned into five protocol phases.

Phase A: residual sending under an established ratchet (10:11:11–10:11:26).

The earliest Ember-tagged event in either logcat is a `DR_SEND_KEY` event on the Pixel 7 Pro at 10:11:11.287 with `msgIdx = 0`, `pn = 0`. This is a sending-side message-key derivation, not a key establishment event; the values indicate that the Pixel 7 Pro is at the very start of a fresh sending chain whose preceding DH ratchet step occurred outside the capture window. Three further `DR_SEND_KEY` events follow at 10:11:11.287, 10:11:15.006, and 10:11:26.217, with `msgIdx` incrementing from 0 to 2 and `pn` held constant at 0. This is the textbook signature of a pure symmetric chain advance during a unidirectional sending burst: the Double Ratchet's KDF chain is being driven forward without a Diffie–Hellman step, exactly as specified in the protocol design ([Section 5.3](#)). No handshake-initiation or handshake-completion event appears prior to this point; the conversation already existed at the moment logcat capture began, and the X3DH handshake that originally established it occurred outside the capture window.

Phase B: first observed DH ratchet step (10:11:32).

The first inbound message of the session arrives on the Pixel 7 Pro at 10:11:32.051, signalled by a `YGGDRASIL_PEER_CONNECTED_INBOUND` event followed by listener and bridge entries. Within 53 ms of the inbound connection's acceptance, the canonical Double Ratchet receive-side sequence is logged: a `DR_RECV_DH_TRIGGERED` event at 10:11:32.087, a `DR_DH_RATCHET_STEP` at 10:11:32.097, and a `DR_RECV_KEY` with `msgIdx = 0`, `pn = 1`, `dhRatchet = true` at 10:11:32.104. The receiver detects that the remote sender's ephemeral DH public key has changed, performs the ratchet step and then derives the message key for the newly arrived message at index zero of the new receiving chain. In order to

perform the ratchet step, the receiver computes a new Diffie–Hellman shared secret, derives a new root key and chain keys, generates a fresh local DH key pair, and discards the previous sending chain. The $pn = 1$ value carried in the message indicates that the remote sender had transmitted exactly one message in its previous sending chain before performing this DH ratchet, which is internally consistent with the Pixel Fold having been quiescent during Phase A and replying for the first time. The receive-side DH ratchet itself completes in approximately 17 ms (the gap between `DR_RECV_DH_TRIGGERED` and `DR_RECV_KEY`); the remaining 36 ms of the 53 ms total covers envelope parsing, bridging, and HMAC verification.

Phase C: alternating sustained exchanges (10:11:32–10:13:47).

After the first direction reversal, the session settles into an alternating bidirectional pattern in which each direction reversal triggers a fresh DH ratchet step on the receiving side. The complete Double Ratchet event sequence captured during the session is presented in [Table 5](#) in chronological order across both devices.

Table 5: Complete chronological sequence of double ratchet events captured across both devices.

Time (UTC)	Device	Event	msgIdx/pn	Notes
10:11:11.287	P7	DR_SEND_KEY	0/0	Start of sending chain (ratchet pre-existing)
10:11:15.006	P7	DR_SEND_KEY	1/0	Symmetric chain advance
10:11:26.217	P7	DR_SEND_KEY	2/0	Symmetric chain advance
10:11:32.087	P7	DR_RECV_DH_TRIGGERED	—	First reply from peer detected
10:11:32.097	P7	DR_DH_RATCHET_STEP	—	DH step performed; old chain discarded
10:11:32.104	P7	DR_RECV_KEY	0/1	dhRatchet = true; first msg of new rcv chain
10:11:45.103	P7	DR_RECV_KEY	1/1	dhRatchet = false; symmetric rcv advance
10:11:48.404	P7	DR_SEND_KEY	0/3	New sending chain; pn = 3 records prior chain length
10:11:56.419	P7	DR_SEND_KEY	1/3	Symmetric chain advance
10:12:06.570	FD	DR_SEND_KEY	0/2	Fold begins a new sending chain
10:12:06.751	P7	DR_DH_RATCHET_STEP	—	P7 performs DH step on receipt
10:12:06.763	P7	DR_RECV_KEY	0/2	dhRatchet = true
10:12:13.952	P7	DR_SEND_KEY	0/2	P7 starts a new sending chain
10:12:14.109	FD	DR_DH_RATCHET_STEP	—	Fold performs DH step on receipt
10:12:14.120	FD	DR_RECV_KEY	0/2	dhRatchet = true
10:12:42.889	FD	DR_SEND_KEY	0/1	Fold sending chain
10:12:43.323	P7	DR_DH_RATCHET_STEP	—	DH step
10:12:43.340	P7	DR_RECV_KEY	0/1	dhRatchet = true
10:13:39.305	P7	DR_SEND_KEY	0/1	P7 sending chain
10:13:39.729	FD	DR_DH_RATCHET_STEP	—	DH step
10:13:39.740	FD	DR_RECV_KEY	0/1	dhRatchet = true
10:13:47.358	FD	DR_SEND_KEY	0/1	Fold sending chain
10:13:47.547	P7	DR_DH_RATCHET_STEP	—	DH step
10:13:47.560	P7	DR_RECV_KEY	0/1	dhRatchet = true

Two structural properties of the Double Ratchet are evidenced by [Table 5](#) with no exceptions. First, every direction reversal triggers a `DR_DH_RATCHET_STEP` event on the receiving side. Of the eight `DR_RECV_KEY` events recorded, seven carry `dhRatchet = true` (each strictly preceded by a `DR_DH_RATCHET_STEP` event) and one carries `dhRatchet = false`. The single `dhRatchet = false` event at 10:11:45.103 is the second message of a same-direction burst from the Pixel Fold to the Pixel 7 Pro and corresponds to a pure symmetric receiving-chain advance rather than a fresh DH step. This is exactly the sending/receiving chain bifurcation specified in the Double Ratchet design: DH ratchet steps occur only on direction reversal, while same-direction continuation messages are processed by the symmetric KDF chain alone.

Second, the previous-chain length field `pn` is propagated and reset correctly across all direction reversals. When the Pixel 7 Pro begins a new sending chain at 10:11:48.404, it stamps `pn = 3` onto the outgoing message, faithfully reporting the length of the previous P7 sending chain (`msgIdx` 0–2 in Phase A, three messages). When the Fold begins its sending chain at 10:12:06.570, it stamps `pn = 2` onto its outbound message, recording the length of the previous Fold sending chain. These `pn` values are not synthesised by the test harness; they are propagated through the live ratchet state machine and re-emitted at the receiver's `DR_RECV_KEY` event with matching values, confirming that the chain-skipping mechanism is wired through correctly end-to-end.

Phase D: TTL cleanup (continuous, throughout the session).

Each outbound message is accompanied by a `TTL_ASSIGNED_OUTBOUND` event recording the assigned TTL (uniformly 60 s throughout this session) and the absolute `expiresAt` timestamp. The `TtlScheduler` is invoked at the same instant. Approximately 60 s later, a `TTL_CLEANUP_SCAN` event is logged on the originating device, followed immediately by a `TTL_CLEANUP_DELETED` event citing the affected message identifier. Table 6 reports the assignment-to-deletion interval for the eight outbound messages whose origins were captured in the logcat window.

Table 6: TTL assignment-to-deletion intervals for outbound messages whose origin was captured.

ID Prefix	Assigned	Deleted	Δ (s)	Excess
554e743f	10:11:11.299	10:12:11.337	60.04	+38 ms
89d7fd0d	10:11:15.017	10:12:15.067	60.05	+50 ms
5d4f2a61	10:11:26.246	10:12:26.344	60.10	+98 ms
2c90ca0b	10:11:48.428	10:12:48.500	60.07	+72 ms
2daa2355	10:11:56.457	10:12:56.506	60.05	+49 ms
57ec8d54	10:12:06.578	10:13:06.669	60.09	+91 ms
dd5982b4	10:12:13.975	10:13:14.023	60.05	+48 ms
db79df65	10:12:42.911	10:13:42.964	60.05	+53 ms

The mean assignment-to-deletion interval is 60.06 s, with a maximum of 60.10 s and a minimum of 60.04 s. Across all eight observed messages the cleanup latency is bounded by approximately 100 ms above the nominal TTL. This is considerably tighter than the bound that would be expected from a coarse periodic-sweep cleanup model, and indicates that the TTL scheduler is using targeted delayed jobs anchored at each message's `expiresAt` timestamp rather than a long-period background sweep. The result is a strong empirical demonstration of Ember's application-level data minimisation property: ciphertext records exit the encrypted database within a tight, predictable bound of their nominal expiry, with no observed exceptions or failures. The forensic significance of this behaviour are examined in Section 7.5.6.

Phase E: pre-key bundle republication (10:12:21).

Exactly one X3DH-related event is observed in the entire session: an `X3DH_PREKEY_BUNDLE_PUBLISHED` event on the Pixel Fold at 10:12:21.492. This event is emitted approximately seventy seconds into the session, at a moment when the active ratchet state for the existing conversation is unchanged and unaffected. No corresponding handshake events are logged on either device. The most plausible interpretation, consistent with the surrounding context, is that this is a routine pre-key bundle republication. Therefore, the Fold is making fresh pre-keys available for any future X3DH handshake initiated against it by another peer, and is unrelated to the conversation currently in progress. As a consequence, the captured session does not exercise an X3DH handshake end-to-end; the captured conversation pre-existed the test

window. The X3DH handshake is therefore described from the implementation in [Sections 6 and 7](#), with empirical validation deferred to future work in which a fresh conversation is established within the capture window ([Section 9](#)).

7.5.4 Yggdrasil Link-Layer Traffic Characterisation

Within the messaging window (10:11:11–10:13:47), the packet capture contains 1493 TCP frames. Of these, 885 frames belong to a single direct TCP exchange between the link-local IPv6 interfaces of the phones and the laptop relay on the shared subnet; this is the encrypted Yggdrasil link-layer traffic. The remainder of the TCP frames in the window are unrelated host activity on the laptop and are excluded from the analysis.

The 885 link-layer frames decompose into 103 distinct TCP flows, each identified by a unique source-port/destination-port pair. Every flow is initiated in the same direction and exhibits an essentially identical packet structure: a SYN, a SYN-ACK, an ACK to complete the handshake, a single PSH-ACK data segment carrying exactly 1479 bytes of TCP payload, and a teardown via FIN-ACK or RST. [Table 7](#) summarises the distribution.

Table 7: Aggregate properties of the 103 Yggdrasil link-layer TCP flows observed during the messaging window.

Property	Value
Distinct TCP flows in window	103
Direction of TCP-SYN initiator	Always the same peer (100% of flows)
Median flow duration	17.3 ms
Mean flow duration	21.9 ms
Maximum flow duration	114.5 ms
Median TCP payload per flow	1479 bytes
Maximum TCP payload per flow	1479 bytes
Variance in payload size	Zero
Median inter-flow arrival	1477 ms
Mean inter-flow arrival	1528 ms

A direct correlation analysis was performed between the start times of these 103 link-layer flows and the eleven Ember message-send events recorded in the logcat trace. For each Ember send event, the closest link-layer flow start within ± 500 ms was identified; many Ember sends had no associated flow within that window at all, and those that did exhibited apparently arbitrary offsets ranging from -442 to $+377$ ms with no consistent pattern. The 1.5-s flow cadence persists at uniform intervals throughout the entire window irrespective of whether Ember is sending, receiving, or idle, including during the 56-s quiescent gap between 10:12:43 and 10:13:39 in which no application-level message is exchanged.

Two conclusions follow. First, the captured TCP flows are not Ember message envelopes; they are Yggdrasil's own link-layer routing and keepalive traffic, which Yggdrasil generates regardless of upper-layer application activity. Second, and more importantly for the security claims of the paper, this means that an observer at the laptop relay vantage point, which is exactly the on-path adversary contemplated in Ember's threat model, sees a uniform, machine-generated, fixed-size, fixed-cadence traffic profile that is structurally decoupled from any actual messaging activity. Inferring that an Ember message has been sent at any specific moment, or that any messages have been sent at all during the session, is not possible from the visible traffic. The constant 1479-byte payload is the result of Yggdrasil padding all link-layer frames to its configured MTU, which provides an unintentional but useful form of fixed-size message obfuscation against this category of

adversary. This property holds against a passive observer co-located with a relay node but does not hold against a global passive adversary or against an adversary with simultaneous visibility into both phones' wireless interfaces.

An exhaustive byte-level scan of the link-layer flow payloads was performed, searching for: any occurrence of the conversation identifier in either ASCII or canonical binary form; any occurrence of the device display names or message UUIDs assigned in `TTL_ASSIGNED_OUTBOUND` events; any JSON envelope structure characteristic of the Ember envelope format; and any occurrence of the byte patterns corresponding to the two peers' Yggdrasil overlay-IPv6 addresses. None of these searches returned any matches at any position in any flow (see [Table 8](#)). The absence of overlay-IPv6 addresses in the wire data, despite their being clearly present in the application logs, is the most direct empirical confirmation that Yggdrasil's link layer encrypts and encapsulates the overlay addresses themselves rather than transmitting them in cleartext within an outer header. From a relay-adjacent vantage point, an observer cannot determine which Yggdrasil overlay nodes are conversing, and only that two adjacent Yggdrasil link-layer peers are exchanging traffic.

Table 8: Byte-level wire scan of Yggdrasil link-layer flow payloads.

Search Target	Match in Capture
Conversation identifier (ASCII/binary)	None found
Device display names	None found
Message UUIDs (from TTL events)	None found
Ember JSON envelope structure	None found
Peer overlay-IPv6 addresses (200::/8, 202::/8)	None found

7.5.5 End-to-End Latency Decomposition

Two latency decompositions can be computed directly from the logcat data. The first is the time taken to establish the outbound TCP connection to the peer over Yggdrasil, measured from the `YGGDRASIL_SEND_START` event to the matching `YGGDRASIL_CONNECTED` event on the same device. The second is the end-to-end delivery latency from the `DR_SEND_KEY` event on the sender to the corresponding `DR_RECV_KEY` event on the receiver, matched by `msgIdx` and `pn`. [Table 9](#) reports the results.

Table 9: Latency decomposition for the dynamic test session.

Stage	N	Min	Median	Mean	Max
Yggdrasil TCP connect (<code>SEND_START</code> → <code>CONNECTED</code>)	10	31 ms	161 ms	144 ms	230 ms
End-to-end (sender <code>DR_SEND_KEY</code> → receiver <code>DR_RECV_KEY</code>)	5	168 ms	202 ms	290 ms	451 ms

Only five end-to-end pairs are computable from the trace: the remaining message events represent either Pixel 7 Pro sends preceding the Fold's first appearance in the logcat (Phase A) or messages whose sender-side `DR_SEND_KEY` events fell outside the captured window. The five computable pairs span both directions (three from P7 to FD and two from FD to P7) and exhibit a median end-to-end latency of 202 ms. The additional cost of the DH ratchet step on every direction reversal is small enough that it is not separately distinguishable in the end-to-end measurements: the receive-side DH ratchet completes in approximately 17 ms in every observed instance and is dominated entirely by network transit and TCP connection establishment. The per-message TCP connection cost remains the dominant contributor to

end-to-end latency, with a median of 161 ms accounting for the majority of the 202 ms median end-to-end figure. The introduction of the Double Ratchet has therefore not measurably worsened Ember's latency profile relative to a hypothetical static-key baseline; cryptographic processing overhead is bounded well below the dominant per-connection TCP establishment cost.

7.5.6 Forward Secrecy and the TTL Cleanup Gap

The Double Ratchet behaviour reported in [Section 7.5.3](#) provides direct positive evidence for the structural property on which forward secrecy depends: that the receiving party performs a Diffie–Hellman ratchet step on every direction reversal and therefore generates a fresh sending chain whose keys are not derivable from any earlier chain state without knowledge of the corresponding fresh DH private key. The seven observed `DR_DH_RATCHET_STEP` events confirm that the protocol is in fact ratcheting, and not silently degenerating into a static-key mode. The TTL-triggered deletion of message records is similarly evidenced: every one of the eight outbound messages whose assignment was captured was deleted from the local database within approximately 100 ms of its nominal expiry, with no observed exceptions or failures.

A more nuanced finding qualifies these positive results. Every `TTL_CLEANUP_DELETED` event in the captured trace carries a structured suffix appended by the instrumentation itself, of the form `note=MessageKeyStore_entries_NOT_purged_here`. This note appears verbatim on every cleanup event throughout the session, on both devices, without exception. Read literally, it states that the TTL cleaner removes the ciphertext record for an expired message from the encrypted message database but does not remove the corresponding entry from the `MessageKeyStore`, the component that holds the per-message symmetric keys, including any skipped message keys retained to allow out-of-order delivery within a chain.

This is a meaningful gap in the forward-secrecy story. The Double Ratchet's forward-secrecy guarantee depends on two complementary operations: deriving a one-time message key from the chain key, and then immediately deleting that one-time key once it has been consumed or once the associated message is no longer addressable. If the per-message key persists in stable storage for the full lifetime of the application, even after the corresponding ciphertext has been deleted, then a forensic adversary who later compromises the device can recover keys for messages the user has been led to believe are no longer recoverable. Application-layer deletion of the ciphertext record is necessary but not by itself sufficient to deliver the ephemerality property the system aims to provide. Ember's TTL mechanism therefore enforces minimisation of ciphertext records but, in its current implementation, does not symmetrically minimise the corresponding message-key state.

We treat this as an identified implementation gap rather than a fundamental design limitation: the TTL cleaner can be extended to invoke a corresponding `MessageKeyStore.purge(messageId)` call atomically with the ciphertext deletion, restoring symmetry between the two operations and closing the residual forensic-recovery surface. The explicit instrumentation note suggests that the gap is already known to the implementer and is anticipated as future work rather than a designed-in property. Until that change is in place, Ember's ephemerality and forward-secrecy claims must be read with this qualification: forward secrecy holds against an adversary who compromises key state *after* the relevant chain keys have been ratcheted past and the message keys purged, but the current failure to purge message keys on ciphertext expiry weakens the guarantee against a forensic adversary who images the device while stale key entries remain. This qualification is carried explicitly into the limitations ([Section 9](#)).

7.5.7 Instrumentation Caveat and Production Implications

The test build used for this evaluation intentionally logs structured cryptographic-pipeline events at fine granularity to enable precise correlation between user actions and network behaviour. These diagnostic

log entries confirm the end-to-end path from user input through Double Ratchet key derivation, network transmission, HMAC verification, ratchet state advance on the receiver, and AEAD decryption. The instrumentation does not log message plaintext, but it does emit ratchet state fingerprints and key indices that would represent a sensitive metadata leak if present in a production build.

It is essential to note that any release build of Ember must suppress this debug instrumentation entirely; Android logcat output is accessible to processes with appropriate permissions and may persist in system log buffers. The instrumented build used here is appropriate only for controlled testing environments. This observation reinforces the broader principle that transport-layer and application-layer encryption are necessary but not sufficient conditions for end-to-end confidentiality: endpoint hygiene, including suppression of diagnostic artefacts, is equally critical.

7.5.8 Synthesis of Dynamic Testing Evidence

The dynamic test conducted on 7 April 2026 provides direct empirical support for the following claims, summarised in [Table 10](#). The Double Ratchet protocol is correctly wired into the live message-processing pipeline: direction-reversal-triggered DH ratchet steps occur reliably, the previous-chain-length field `pn` is propagated and reset correctly across chain transitions, and same-direction continuation messages are processed via pure symmetric chain advances without redundant DH operations. The verify-then-decrypt processing order is upheld for every inbound message; HMAC verification precedes any key derivation in every observed instance. Cryptographic processing overhead is small relative to network transit, with the receive-side DH ratchet completing in approximately 17 ms per direction reversal. The Yggdrasil link layer encrypts and encapsulates overlay-IPv6 addresses such that they are not visible to a relay-adjacent on-path observer, and the visible link-layer traffic profile is uniform, fixed-size, fixed-cadence, and structurally decoupled from Ember message activity. The TTL cleanup mechanism reliably deletes expired ciphertext records within approximately 100 ms of nominal expiry.

Table 10: Summary of dynamic testing evidence and supported security claims.

Security Property	Evidence Source	Verdict
Double Ratchet active (FS structure)	Logcat: 7 DH ratchet steps, one per direction reversal	Confirmed
Chain-skipping (<code>pn</code>) correctness	Logcat: <code>pn</code> propagated/reset across all reversals	Confirmed
Verify-then-decrypt ordering	Logcat: HMAC verified before ratchet/decrypt on all msgs	Confirmed
Symmetric chain advance (no DH)	Logcat: same-direction msg has <code>dhRatchet = false</code>	Confirmed
TTL ciphertext deletion	Logcat: 8/8 deleted within ~100 ms of expiry	Confirmed
Overlay address concealment	PCAP: no <code>200::/8</code> , <code>202::/8</code> frames; wire scan 0 matches	Confirmed
Link-layer metadata bounded	PCAP: uniform 1479-byte, fixed-cadence, msg-decoupled	Confirmed (relay-adjacent)
Per-message-key destruction	Logcat: <code>MessageKeyStore</code> not purged on cleanup	Gap identified (Section 7.5.6)
X3DH handshake end-to-end	Conversation pre-existed capture window	Not exercised (future work)

Conversely, the test does not establish, and the paper does not claim, validation of the X3DH handshake itself, since the captured session does not exercise an X3DH handshake end-to-end (the conversation pre-existed the capture window). Nor does the test establish forward secrecy in the strong sense, in light of the `MessageKeyStore` retention behaviour reported in [Section 7.5.6](#). Finally, the relay-adjacent capture vantage point cannot rule out plaintext leakage at other points in the data path; a stronger plaintext-non-recovery claim would require a complementary capture taken at the Yggdrasil tunnel interface on one of the phones. Within these bounds, the dynamic test materially strengthens the empirical evidence base for Ember 4.0's core design claims and identifies one specific implementation gap whose remediation is straightforward.

7.6 Threat Model Coverage and Residual Risk

Threat modelling considered eight realistic threat actors spanning passive observation, active network manipulation, compromised peers, compromised devices, metadata-focused adversaries, timing/side-channel specialists, replay attackers, and high-capability cryptanalysts. Mitigations were mapped per actor and a residual-risk classification was produced as shown in [Table 11](#).

Table 11: Threat actor assessment summary (probability, impact, and residual risk).

Threat Actor	Probability	Impact	Residual Risk
Passive eavesdropper	High	Low	Very Low
Active MITM attacker	Medium	Medium	Low
Compromised contact (known key)	Low–Medium	High	Medium
Compromised device (full access)	Low	Critical	High
Forensic adversary (post-expiry)	Low	High	Medium
Metadata collector (social graph)	Medium	Low	Low–Medium
Replay attacker	Medium	Low	Very Low
Cryptanalysis expert	Very Low	Critical	Very Low

Dominant residual risks.

The highest residual risk is endpoint compromise (root/debuggable device or active process compromise), which is a dominant failure mode for mobile secure messaging generally. Ember's design reduces the forensic value of persistent storage via ciphertext-only persistence and SQLCipher, and the Double Ratchet bounds the temporal window of a transient compromise via post-compromise security; but neither can fully mitigate a hostile runtime environment with memory inspection or UI subversion. A distinct and newly itemised residual risk is the *forensic adversary who images the device after message expiry*: because the `MessageKeyStore` is not purged on TTL cleanup ([Section 7.5.6](#)), retained per-message keys expand the recovery surface for expired messages until that gap is closed. The next most salient residual risks are metadata inference via traffic analysis by an adversary stronger than the relay-adjacent observer empirically characterised here, and timing/side-channel uncertainty (not empirically tested in this phase).

7.7 Summary of Evidence and Confidence

Across static analysis, unit-test validation, ASVS mapping, threat modelling, and dynamic network security analysis, the overall security posture is assessed as **strong for content security** (confidentiality/integrity/authentication) with **empirically demonstrated ratcheting behaviour**, and with **bounded claims** for metadata privacy, forward secrecy against a forensic adversary, and side-channel resilience. The confidence level is **high** for correctness of the cryptographic composition and the live operation of the

Double Ratchet (supported by code review, unit tests, and runtime telemetry confirming direction-reversal-triggered DH ratchet steps with correct `pn` propagation), **medium-high** for storage protections and TTL minimisation of ciphertext, and **medium** for metadata privacy and side-channel resilience due to the relay-adjacent capture vantage and the absence of fuzzing and timing measurement. The `MessageKeyStore` retention gap is the single most actionable item lowering the forward-secrecy confidence against a forensic adversary (see Table 12).

Table 12: Evidence-backed security metrics (paper-ready summary).

Metric	Value
Overall security assessment (static + dynamic)	4.5/5.0
ASVS alignment (assessed subset)	77% (L2 partial L3)
Crypto unit tests	7/7 passed (100%)
Encryption mean (1 KB, unit-test)	3.2 ms
Decryption mean (1 KB, unit-test)	2.9 ms
Static-analysis high/critical findings	1/0
Static-analysis findings (total)	12 (1 high, 3 medium, 4 low, 4 info)
Dynamic: DH ratchet steps observed	7 (one per direction reversal)
Dynamic: <code>DR_RECV_KEY</code> with <code>dhRatchet = true</code>	7/8
Dynamic: HMAC verify-then-decrypt upheld	All inbound messages
Dynamic: receive-side DH ratchet latency	~17 ms
Dynamic: median end-to-end latency	202 ms
Dynamic: median Yggdrasil TCP connect	161 ms
Dynamic: TTL cleanup bound above expiry	~100 ms (mean Δ 60.06 s)
Dynamic: overlay addresses recovered from wire	0
Identified implementation gap	<code>MessageKeyStore</code> not purged on TTL cleanup

8 Evaluation and Discussion

This section evaluates Ember against its stated research objectives and design goals. The evaluation draws on three complementary evidence sources: functional correctness observations from the dynamic test session reported in Section 7.5, performance measurements decomposed from correlated device telemetry, and qualitative analysis of the design trade-offs inherent to Ember’s architectural decisions. The discussion is structured to make claims proportionate to evidence, and to identify precisely where confidence is strong, where it is bounded, and where further work is required.

8.1 Functional Validation

Functional validation assesses whether Ember’s implemented behaviour corresponds to the security invariants and processing pipelines specified in Sections 4–6. Rather than relying solely on unit tests (which validate individual components in isolation), this evaluation draws on the dynamic test session to confirm that the integrated system behaves correctly under real operating conditions.

End-to-end ratcheting behaviour.

The central functional result is that the Double Ratchet operates correctly end-to-end in the live system. Across the captured session, every direction reversal triggered a receive-side DH ratchet step, every same-direction continuation message was processed by a pure symmetric chain advance, and the previous-chain-length field `pn` was propagated from sender to receiver with matching values at every transition (Section 7.5.3, Table 5). This confirms that the chain bifurcation and chain-skipping mechanisms

specified in [Section 5.3](#) are wired through the live state machine rather than merely present in dormant code. The deterministic conversation identifier was correctly resolved on both devices for every inbound and outbound message, confirming that the canonicalised hash-based conversation identity mechanism described in [Section 5.1](#) functions as specified.

Cryptographic pipeline correctness.

For each message, runtime telemetry confirms that the sending pipeline executed in the correct order: ratchet-derived message-key generation, AES-256-GCM encryption, HMAC-SHA256 computation over envelope fields (including the ratchet header), ciphertext persistence to the encrypted database, and network transmission of the encrypted envelope. On the receiving side, the pipeline likewise executed correctly: envelope reception via the foreground listener, HMAC verification prior to any ratchet state update or decryption (verify-then-decrypt), ratchet advance, ciphertext persistence, and in-memory decryption for UI rendering. No deviations from the specified processing order were observed. In particular, no message was decrypted prior to successful HMAC verification, and no plaintext was written to persistent storage at any point.

Ciphertext-only persistence invariant.

The telemetry confirms that for every message, the `ChatRepository` persisted the ciphertext record to the SQLCipher-encrypted database before performing decryption. Decrypted plaintext was produced only in-memory for the display pipeline and was not observed in any database write operation. While this evidence does not constitute a formal proof that plaintext is never persisted under all possible execution paths, it provides empirical confirmation that the invariant holds under normal operational conditions across a representative message exchange.

TTL minimisation of ciphertext.

Every outbound message whose origin was captured was assigned a TTL and deleted from the encrypted database within approximately 100 ms of its nominal expiry ([Section 7.5.3](#), [Table 6](#)). This confirms that the `WorkManager-plus-targeted-delayed-jobs` expiration mechanism described in [Section 8.3](#) is correctly integrated and operates with tight, predictable latency. As discussed in [Section 7.5.6](#), this minimisation applies to ciphertext records but not, in the current implementation, to the corresponding per-message keys in the `MessageKeyStore`, a gap that bounds the strength of the ephemerality guarantee against a forensic adversary.

Network transport correctness.

The `YggdrasilClient` successfully validated overlay IPv6 addresses, established TCP connections, and transmitted encrypted envelopes for all outbound messages. Inbound messages were correctly received by the foreground listener service, forwarded through the `MessageBridge`, and processed by the domain layer without errors. The absence of connection failures during the test is encouraging, though it reflects benign network conditions on a local hotspot rather than adversarial or degraded environments.

Summary.

Functional validation confirms that Ember's core security invariants including, per-message ratcheted authenticated encryption, verify-then-decrypt processing, ciphertext-only persistence, and TTL minimisation of ciphertext, are upheld during live operation across a representative message exchange. The system delivered all messages correctly, maintained cryptographic pipeline integrity, exhibited correct Double Ratchet behaviour at every direction reversal, and revealed one specific implementation gap (`MessageKeyStore` retention) that is documented transparently rather than elided.

8.2 Performance Characteristics

Performance evaluation decomposes end-to-end message latency into constituent stages using timestamps extracted from correlated logcat telemetry on both devices ([Section 7.5.5](#), [Table 9](#)). This decomposition allows identification of the dominant contributors to delivery latency and provides a basis for assessing whether Ember's performance is compatible with interactive messaging use cases.

8.2.1 Cryptographic Overhead Including the Ratchet

Application-layer cryptographic processing now comprises ratchet-derived message-key generation, AES-256-GCM encryption, HMAC-SHA256 computation, and a Diffie–Hellman ratchet step. The dynamic test isolates the cost of the most expensive of these, the receive-side DH ratchet, at approximately 17 ms per direction reversal (measured as the interval between `DR_RECV_DH_TRIGGERED` and `DR_RECV_KEY`). Same-direction continuation messages incur no DH operation at all and are processed by the symmetric KDF chain alone. These figures are consistent with the unit-test micro-benchmarks of [Table 3](#), in which HKDF derivation and HMAC generation each complete in under 2 ms and AES-GCM encryption/decryption in approximately 3 ms for 1 KB payloads.

The key performance finding is that the introduction of the Double Ratchet has not measurably worsened Ember's latency profile relative to a hypothetical static-key baseline. The 17 ms DH ratchet cost is more than an order of magnitude smaller than the dominant per-connection TCP establishment cost (median 161 ms), and is not separately distinguishable in the end-to-end measurements. Cryptographic processing therefore remains comfortably within the latency budget for interactive text messaging, and would remain acceptable even on lower-end devices where hardware-backed AES acceleration may be less performant.

8.2.2 Network Transport Latency

The dominant contributor to end-to-end message delivery latency is TCP connection establishment through the Yggdrasil overlay. For each outbound message, the `YggdrasilClient` opens a new TCP connection to the peer's overlay IPv6 address. Across the ten measured connections, establishment latency had a median of 161 ms and a mean of 144 ms, ranging from 31 ms (best case, suggesting reuse of overlay routing state) to 230 ms. The resulting end-to-end latency, from sender-side message-key derivation to receiver-side message-key derivation, had a median of 202 ms across the five computable pairs, with the per-connection TCP cost accounting for the majority of that figure.

These steady-state latencies are competitive with centralised messaging systems that typically achieve 100–500 ms delivery latency under favourable conditions, and confirm that Ember's transport model supports interactive messaging without prohibitive delay. The per-connection model adopted by Ember (one TCP connection per message send) is intentionally conservative, prioritising simplicity and analysability over connection pooling optimisation. As discussed below and in [Section 9](#), persistent connection pooling could reduce mean delivery latency by amortising connection establishment costs across multiple messages, at the cost of additional state-management complexity.

8.2.3 Consistency with Unit-Test Benchmarks

The on-device measurements are broadly consistent with the unit-test micro-benchmarks reported in [Section 7.3](#), with expected differences attributable to database I/O, Room persistence overhead, and real-network transport delays that are absent in isolated unit tests. This consistency provides additional confidence that the unit-test benchmarks are representative of actual system behaviour, albeit with the caveat

that the dynamic test was conducted under favourable network conditions (local hotspot, low contention) that may not reflect all deployment scenarios.

8.3 Design Trade-Offs

Ember's architecture reflects a series of deliberate trade-offs between security properties, system complexity, usability, and deployability. This subsection analyses the most consequential of these trade-offs, evaluating both the benefits obtained and the costs incurred.

8.3.1 Serverlessness vs. Availability

The decision to operate without central servers eliminates metadata aggregation points, provider-controlled storage, and regulatory compellability risks associated with centralised architectures. However, it imposes a fundamental availability constraint: message delivery requires both peers to be simultaneously reachable on the overlay network. Unlike centralised systems that queue messages for offline recipients, Ember cannot deliver messages to peers that are not currently connected. This trade-off is explicitly bounded in the threat model ([Section 3](#)) and is consistent with the design philosophy of Ricochet and early Briar, which accept similar availability limitations in exchange for reduced infrastructure trust [[12,14](#)].

The serverless posture interacts directly with the choice of key-establishment protocol. Because Ember has no pre-key server, the X3DH handshake that bootstraps each conversation must currently occur while both peers are reachable, rather than asynchronously against a published pre-key bundle. The dynamic test did not exercise this handshake (the conversation pre-existed the capture window; [Section 7.5.3](#)), so its end-to-end behaviour under live conditions remains a validation gap rather than a demonstrated property. Evaluating Ember's behaviour under degraded conditions, including overlay partition, asymmetric reachability, and Android background execution throttling, remains an important direction for future work.

8.3.2 Integrating the Double Ratchet into a Serverless Architecture

The most consequential cryptographic decision in this version of Ember is the activation of a full X3DH-plus-Double-Ratchet stack in place of the earlier static per-conversation keys with manual rotation. This delivers per-message forward secrecy and post-compromise security but it does so within a serverless architecture that lacks the coordinating infrastructure on which Signal-style deployments ordinarily rely. Several aspects of this integration warrant discussion.

First, the dynamic evaluation confirms that the ratchet operates correctly under realistic message ordering: direction reversals reliably trigger DH ratchet steps, and the `pn` chain-skipping field is propagated and reset correctly across all transitions ([Section 7.5.3](#)). This is a non-trivial result, because the principal risk in porting a ratchet into a peer-to-peer environment is silent degeneration into a static-key mode or desynchronisation under reordering. The seven observed DH ratchet steps, each strictly preceded by a `DR_DH_RATCHET_STEP` event and matched to a direction reversal, provide direct evidence that the protocol is genuinely ratcheting.

Second, the performance cost of the ratchet is small. The receive-side DH step completes in approximately 17 ms and is dominated entirely by network transport ([Section 8.2](#)). The earlier concern that ratcheting complexity might impose a prohibitive latency or state-management burden in a serverless setting is not borne out by the measurements: the ratchet's cost is negligible relative to per-connection TCP establishment.

Third, and most importantly, the integration is not yet complete. The dynamic analysis surfaced a concrete gap. The `MessageKeyStore` is not purged when the corresponding ciphertext is deleted on TTL expiry ([Section 7.5.6](#)), meaning that the forward-secrecy guarantee is weaker against a forensic adversary

than the ratchet structure alone would suggest. This is a remediable implementation defect rather than an architectural limitation, but it illustrates a general lesson: achieving forward secrecy requires not only correct key *evolution* but also disciplined key *destruction*, and the two must be wired together. Ember's TTL mechanism currently minimises one half of this pair (ciphertext) without the other (message keys). Closing this gap is identified as the highest-priority remediation item ([Section 9](#)).

The cost of adopting the ratchet, relative to the simpler static-key design it replaces, is a larger and more intricate cryptographic state machine whose correctness is harder to establish by inspection alone. Ember mitigates this through the fine-grained instrumentation that made the dynamic validation possible; the same instrumentation, however, must be suppressed in release builds to avoid leaking ratchet-state metadata. The trade-off is therefore favourable but conditional: the ratchet provides materially stronger guarantees at negligible performance cost, provided that the key-destruction gap is closed and that diagnostic instrumentation is stripped from production builds.

8.3.3 Ciphertext-Only Persistence vs. Usability

Ember's decision to store only ciphertext in the local database and to decrypt messages exclusively in-memory for UI rendering provides strong protection against disk-level data extraction. However, this design imposes a performance cost: every time a conversation view is opened, all visible messages must be decrypted from the database. For conversations with large message histories, this could introduce perceptible latency in the display pipeline. With ratcheting now in place, display-path decryption additionally depends on the availability of the corresponding per-message keys in the `MessageKeyStore`; this is the same store whose retention behaviour creates the forward-secrecy gap, which highlights a genuine tension between displayability of recent history and aggressive key destruction. Potential mitigations include paginated decryption, bounded conversation history aligned with the TTL policy, and a key-retention window that is short but non-zero. Each approach must be evaluated against Ember's plaintext-minimisation goals to ensure that caching does not inadvertently extend plaintext or key lifetime beyond acceptable bounds.

8.3.4 Foreground Service vs. Battery and Usability

Ember's use of a foreground service for inbound message reception is the only mechanism available on modern Android to maintain a persistent TCP listener without relying on third-party push notification infrastructure. This design choice preserves metadata privacy by avoiding FCM or APNs, but at the cost of increased battery consumption and a persistent notification that may be perceived as intrusive by users. The trade-off is defensible within Ember's threat model: push notification infrastructure has been empirically shown to leak sensitive metadata in a majority of messaging applications [23], and avoiding this leakage vector is a core design goal. However, the battery and usability costs may limit Ember's practical adoption beyond privacy-conscious users. Future work could explore hybrid approaches, such as low-frequency polling, wake-on-overlay signalling, or negotiated delivery windows. All these options could reduce continuous resource consumption while preserving the avoidance of third-party push infrastructure.

8.3.5 Explicit Envelope HMAC vs. Redundancy

Ember's decision to compute an explicit HMAC-SHA256 over envelope fields in addition to the AES-GCM authentication tag introduces a degree of cryptographic redundancy. AES-GCM already provides authenticated encryption, and under correct implementation the GCM tag alone would suffice to detect tampering of the ciphertext. The justification for this redundancy is primarily architectural. The explicit HMAC layer enables a strict verify-before-decrypt processing model in which envelope integrity, including

the integrity of the Double Ratchet header fields `dhPub`, `msgIdx`, and `pn`, can be assessed before any ratchet state update or decryption primitive is invoked. This separation provides a clear, auditable boundary: the HMAC gate rejects unauthenticated envelopes before any cryptographic state is advanced, which is particularly important once ratcheting is in play, since processing a forged header could otherwise perturb ratchet state. The dynamic test confirmed that this ordering holds for every inbound message (Section 7.5). The cost is a modest increase in computational overhead (approximately 1.4 ms per message for HMAC generation) and a small increase in envelope size, which is negligible relative to the per-message latency budget.

8.3.6 Summary of Trade-Off Posture

Across these trade-offs, Ember consistently prioritises correctness, analysability, and reduced infrastructure trust over maximal performance or feature breadth. The activation of the Double Ratchet shifts the system meaningfully towards stronger theoretical guarantees while preserving this posture: the ratchet is empirically demonstrated to operate correctly at negligible performance cost, and the one integration gap it surfaced (`MessageKeyStore` retention) is documented, bounded, and straightforward to remediate. The trade-offs are intentional, documented, and reversible: future iterations can selectively relax constraints (e.g., by introducing connection pooling or asynchronous pre-key relays) as correctness is established and the threat model evolves.

9 Limitations and Future Work

This section explicitly documents the limitations of the current Ember implementation and identifies directions for future work that would strengthen its security guarantees, extend its functional scope, and improve confidence in its correctness. Limitations are stated precisely and linked to the relevant design decisions and threat model constraints discussed in preceding sections. Future work items are framed as concrete research and engineering objectives rather than aspirational features, with attention to the prerequisites, risks, and open problems associated with each.

9.1 Current Limitations

9.1.1 Residual Forward-Secrecy Gap: `MessageKeyStore` Retention

While Ember now provides per-message forward secrecy and post-compromise security through the Double Ratchet, the dynamic analysis identified a concrete implementation gap that bounds the forward-secrecy guarantee against a forensic adversary. As reported in Section 7.5.6, the TTL cleaner deletes expired ciphertext records from the encrypted message database but does not symmetrically purge the corresponding per-message keys from the `MessageKeyStore`. Every captured `TTL_CLEANUP_DELETED` event carried the instrumentation note `note = MessageKeyStore_entries_NOT_purged_here`, confirming the behaviour on both devices without exception.

The consequence is that a forensic adversary who images the device after a message has nominally expired may still recover the key required to decrypt that message, even though the ciphertext has been removed from the application database. Forward secrecy in the ratchet sense therefore holds only to the extent that the relevant keys have in fact been deleted, which in the current implementation they are not at ciphertext-expiry time. This is an identified implementation defect rather than a fundamental design limitation: the remediation is to invoke `MessageKeyStore.purge(messageId)` atomically with ciphertext deletion. Until that change is in place, Ember's ephemerality claim should be read as application-level minimisation of ciphertext records rather than as full per-message key destruction, and its forward-secrecy claim should be qualified accordingly.

9.1.2 X3DH Handshake Not Yet Validated End-to-End

The dynamic evaluation validated the Double Ratchet under live conditions but did not exercise the X3DH initial key agreement, because the test conversation pre-existed the capture window (Section 7.5.3, Phase E). The only X3DH-related event observed was a routine pre-key bundle republication unrelated to the active conversation. Consequently, while the X3DH handshake is implemented and described in Sections 6 and 7, its end-to-end behaviour, including the X3DH_HANDSHAKE_INITIATE and X3DH_HANDSHAKE_COMPLETE sequence, the derivation of the initial root key, and the correct seeding of the first sending and receiving chains, has not yet been empirically confirmed under capture. This is a validation gap rather than a known defect, and a dedicated follow-up test (Section 9.5) would close it.

9.1.3 No Group Messaging or Multi-Device Support

Ember currently supports only one-to-one conversations. No group messaging semantics, sender-key distribution, or multi-party state management are implemented. Extending the system to support group communication would require addressing scalable key distribution, membership consistency, and epoch-based key evolution—challenges that are well-characterised by the MLS protocol standardisation effort [10,11] but that introduce substantial complexity in serverless environments where globally consistent state and reliable delivery cannot be assumed.

Similarly, Ember does not support multi-device operation. A user's identity keys, ratchet state, and message history are bound to a single device, and there is no mechanism for synchronising cryptographic state across multiple endpoints. Multi-device support in end-to-end encrypted systems typically requires either a device-linking protocol with cross-device key agreement or a centralised device registry, both of which conflict with Ember's serverless design goals and would interact non-trivially with per-device ratchet state.

9.1.4 Availability Constraints and Offline Delivery

Ember's serverless architecture imposes a synchronous availability requirement: both peers must be simultaneously reachable on the overlay for message delivery to succeed, and, in the absence of a pre-key server, for the initial X3DH handshake to complete. There is no store-and-forward capability, no message queueing infrastructure, and no mechanism for deferred delivery to offline recipients. While this constraint is acceptable for a research prototype and is shared by comparable systems such as Ricochet [12], it limits Ember's practical utility for asynchronous communication patterns that are routine in everyday messaging. The foreground service model mitigates this constraint when the application is running, but Android's increasingly aggressive background execution restrictions may terminate or throttle the listener under certain power-saving states; behaviour under prolonged backgrounding, Doze mode, or OEM-specific process management was not evaluated.

9.1.5 Platform-Specific Storage Remanence

As discussed in Section 2.6 and the threat model (Section 3), Ember cannot guarantee physical secure deletion of expired message data on Android. TTL-driven cleanup removes records from the application database, but residual data may persist in SQLite write-ahead logs, filesystem journal structures, or untrimmed flash storage [19,20]. This limitation is fundamental to the Android storage stack rather than specific to Ember, but it compounds the MessageKeyStore retention gap described above: even once message keys are purged from the application's logical store, residual copies may persist at the storage layer. It bounds the strength of Ember's ephemerality claims to application-level access minimisation rather than forensic-grade erasure.

9.1.6 Test Build Instrumentation as an Endpoint Risk

The dynamic test reported in [Section 7.5](#) relied on an instrumented build that emits fine-grained cryptographic-pipeline events to the Android logcat stream. While this instrumentation does not log message plaintext, it emits ratchet state fingerprints and key indices that would represent a sensitive metadata leak if present in any non-test deployment. The current codebase does not enforce build-variant separation between instrumented and release configurations at the compiler level, meaning that suppression depends on developer discipline rather than structural guarantees. Formalising this separation through build flavour enforcement or compile-time log stripping would reduce the risk of accidental metadata leakage in production builds.

9.1.7 Absence of Side-Channel and Timing Analysis

No empirical measurement of side-channel behaviour has been performed. Timing variations in cryptographic operations, memory access patterns during key handling, and observable differences in processing time between valid and invalid messages could theoretically be exploited by a local or co-located adversary. While the use of standard library implementations of AES-GCM, HMAC-SHA256, and HKDF provides some baseline resistance to timing attacks (assuming constant-time implementations in the underlying Android cryptographic provider), this has not been independently verified for Ember's specific execution environment, and the introduction of data-dependent ratchet branching (DH step on direction reversal vs. symmetric advance) introduces a new potential timing distinction that warrants measurement.

Furthermore, Ember's onboarding trust model operates entirely on a Trust-On-First-Use (TOFU) basis complemented by optional, manual cryptographic fingerprint verification. While this decentralized approach aligns with a serverless architecture devoid of a global identity directory, it fundamentally provides weaker identity guarantees than systems utilizing centralized out-of-band registries or robust third-party verification protocols.

9.2 Future Work: Completing the Ratcheting Implementation

With the Double Ratchet now active and empirically validated for its core behaviour, the highest-priority future work is no longer to introduce ratcheting but to *complete* its integration so that the forward-secrecy guarantee holds against the strongest adversary in Ember's threat model.

Symmetric key destruction on ciphertext expiry.

The single most actionable item is closing the `MessageKeyStore` retention gap ([Section 7.5.6](#)). The TTL cleaner should invoke `MessageKeyStore.purge(messageId)` atomically with the deletion of the corresponding ciphertext record, so that per-message keys, including any skipped message keys retained for out-of-order delivery, are destroyed at the same instant as the ciphertext they protect. This change restores symmetry between ciphertext and key minimisation and closes the residual forensic-recovery surface. A follow-up dynamic test should confirm that, after remediation, no `MessageKeyStore` entry survives the deletion of its associated ciphertext, ideally by asserting the absence of the key entry in a post-expiry database image.

End-to-end validation of the X3DH handshake.

A dedicated follow-up test in which a fresh conversation is established from scratch within the capture window. For example, by reinstalling Ember on one device and re-adding the contact via QR-code exchange would capture the `X3DH_HANDSHAKE_INITIATE` and `X3DH_HANDSHAKE_COMPLETE` events end-to-end and provide direct empirical validation of the initial key agreement, the derivation of the initial root key, and the seeding of the first ratchet chains. This would close the validation gap identified in [Section 7.5.3](#).

Asynchronous key exchange without pre-key servers.

The standard Signal deployment model relies on a central server to publish and distribute pre-key bundles, enabling asynchronous key establishment between parties that are not simultaneously online. Ember's serverless architecture eliminates this infrastructure, so the X3DH handshake currently requires both peers to be reachable. A viable path to asynchronous establishment would need to either (i) restrict initial key exchange to synchronous sessions, (ii) introduce a minimal, untrusted pre-key relay analogous to Cwtch's discardable servers [13], or (iii) explore alternative asynchronous key-exchange constructions that do not depend on persistent public key directories [8]. Each approach entails distinct trade-offs between availability, metadata exposure, and architectural complexity.

State synchronisation and recovery under unreliable delivery.

Ratcheting protocols require that both parties maintain consistent views of the evolving key state. Message loss, reordering, or duplication—all possible in Ember's per-connection TCP model—can desynchronise ratchet state. The dynamic test confirmed correct pn-based chain skipping under benign conditions, but did not exercise loss or reordering. Future work should implement and test robust state-recovery mechanisms, potentially including ratchet-state checkpointing and resynchronisation protocols, while ensuring that such mechanisms do not themselves introduce new attack surfaces. Formal analyses of the Double Ratchet [5–7] demonstrate that security guarantees are tightly coupled across protocol stages and that compositional reasoning is difficult, motivating explicit analysis of the composed protocol (ratchet plus Ember's envelope and authentication layers).

9.3 Future Work: Group Messaging

Extending Ember to support group conversations would significantly broaden its functional utility but introduces challenges that are qualitatively different from those of two-party communication. The MLS protocol [10] provides a well-specified framework for group key management with forward secrecy and post-compromise security, and serves as the primary reference point for any future group messaging extension.

Delivery and consistency semantics.

MLS assumes the availability of a delivery service that can reliably disseminate messages to all group members and maintain a consistent view of group state [10]. In Ember's serverless architecture, no such service exists. A group extension would need to address how messages are distributed to multiple peers (direct fan-out, gossip, or relay-based approaches), how membership changes are propagated consistently, and how the system behaves when some group members are unreachable. Each approach involves trade-offs between latency, bandwidth, consistency, and metadata exposure.

Authentication and identity binding.

Group messaging requires a mechanism for authenticating group membership and binding long-term identities to cryptographic credentials. MLS delegates this to an external authentication service, which is straightforward in centralised deployments but problematic in a fully decentralised environment. Ember would need to develop a peer-to-peer membership management protocol, potentially leveraging its existing key fingerprint verification mechanism (Section 5.7) as a building block for group-level trust establishment.

Scalability and state management.

MLS's tree-based key management structure provides logarithmic scaling for group key updates, but its correctness depends on consistent tree state across all members [11]. Maintaining this consistency without a central coordinator is an open research problem, particularly under network partition or churn. For an initial group extension, Ember might adopt a simpler sender-key model, accepting weaker forward secrecy

properties in exchange for reduced state synchronisation requirements, and migrate to full MLS-style group state as the infrastructure matures.

9.4 Future Work: Formalisation and Verification

The current security assessment of Ember relies on static code analysis, unit testing, OWASP compliance mapping, threat modelling, and dynamic network analysis. While this combination provides multi-source evidence for the system's security properties, it falls short of formal verification. Several directions for formalisation would materially strengthen confidence in Ember's guarantees.

Protocol specification and symbolic analysis.

Ember's composed protocol, which includes X3DH establishment, Double Ratchet evolution, and envelope authentication, could be specified as a formal model and subjected to symbolic analysis using tools such as ProVerif or Tamarin. Such analysis would provide machine-checked guarantees for properties including message confidentiality, authentication, forward secrecy, post-compromise security, and replay resistance under the Dolev–Yao adversary model. Although the ratchet increases the protocol surface relative to the earlier static-key design, the well-studied structure of the Double Ratchet provides a strong starting point for tractable analysis.

Fuzzing and adversarial input testing.

The envelope parser, JSON deserialiser, length-prefix framing handler, and ratchet header processing (`dhPub`, `msgIdx`, `pn`) are all candidates for coverage-guided fuzzing. Fuzzing would complement the existing static analysis by identifying edge cases, boundary-condition failures, and potential denial-of-service vectors that are not detectable through ruleset-driven scanning, and is particularly valuable now that malformed headers could in principle perturb ratchet state if the HMAC gate were ever bypassed.

Side-channel measurement.

Empirical measurement of timing behaviour during cryptographic operations, particularly HMAC verification, AES-GCM decryption, and the data-dependent branch between DH and symmetric ratchet steps, would allow assessment of whether Ember is vulnerable to timing-based oracle attacks. While the use of platform-provided cryptographic primitives offers some baseline protection, the interaction between Ember's application-layer processing and the underlying cryptographic library has not been profiled for timing leakage.

Instrumented device testing.

The performance measurements reported in this paper are derived from logcat timestamps and non-instrumented unit tests. Future work should include Android-instrumented benchmarks executed on physical devices under controlled conditions, measuring cryptographic operation latency (including the DH ratchet step in isolation), database I/O overhead, and end-to-end delivery latency with statistical rigour across a larger message population than the present session afforded.

Adversarial network testing.

The dynamic test was conducted under benign network conditions. Future testing should include adversarial scenarios such as packet injection, replay attacks, connection reset injection, malformed envelope delivery, message reordering and loss (to exercise ratchet resynchronisation), and overlay routing disruption. These tests would validate Ember's defensive processing model under conditions that more closely approximate the active network adversary assumed in the threat model ([Section 3](#)).

9.5 Future Work: Additional Directions

Beyond the primary areas identified above, several additional directions merit consideration.

Connection pooling and persistent sessions.

The per-connection TCP model used by Ember provides simplicity and analysability but incurs connection establishment overhead for every message. As [Section 7.5.5](#) shows, the Yggdrasil TCP connect step has a median cost of 161 ms and accounts for the majority of the 202 ms end-to-end median latency. Introducing persistent connection pooling, where a TCP session is maintained for the duration of a conversation or overlay session, could reduce steady-state delivery latency to values approaching the network transit time alone. The trade-off is increased state management complexity and the need to handle connection interruption gracefully without desynchronising application or ratchet state.

Multimedia and file transfer.

Ember currently supports text-only messaging. Extending the system to support encrypted file or image transfer would require addressing payload size limits, chunked transfer semantics, progress indication, and the interaction between large payloads and the TTL expiration mechanism. The framing protocol's length-prefix design accommodates larger payloads in principle, but practical deployment would need to consider memory constraints on mobile devices and the impact of large transfers on overlay network performance.

Cross-platform deployment.

Ember is currently implemented exclusively for Android. Porting the system to iOS, desktop, or web platforms would broaden its reach but would require re-evaluating platform-specific assumptions around key storage (Android Keystore vs. iOS Keychain vs. platform-specific HSM interfaces), background execution models, and notification infrastructure. A cross-platform implementation would also raise the question of multi-device ratchet-state synchronisation, which is closely related to the group messaging challenge discussed above.

Formal privacy analysis of overlay metadata.

The traffic analysis characterisation reported in [Section 7.5.4](#) provides direct empirical evidence about what a relay-adjacent observer sees, namely a uniform, fixed-size, fixed-cadence link-layer profile decoupled from message activity. A more rigorous analysis would quantify the information leakage available to stronger adversaries, including one with simultaneous visibility into both phones' interfaces or a capture taken at the Yggdrasil tunnel interface, using established metrics from the anonymous communication literature [25,26]. A complementary capture at the tunnel interface would also enable a stronger plaintext-non-recovery claim than the relay-adjacent vantage point used here can support.

Usability evaluation.

Ember's security model places non-trivial responsibilities on users, including manual key fingerprint verification, understanding of TTL semantics, and acceptance of foreground service resource costs. No usability evaluation has been conducted. Future work should assess whether users can correctly perform trust establishment, interpret ratchet and key-evolution events, and understand the boundaries of Ember's ephemerality guarantees. Prior work on user perceptions of deletion in messaging [22] suggests that misalignment between user expectations and actual system behaviour is a significant risk that warrants empirical investigation.

9.6 Discussion

The security assessment combined five complementary evaluation methods: static code analysis (zero critical and one high-severity finding, the latter confined to a ratchet-superseded code path, across 33 files),

unit-test validation of the full cryptographic pipeline (7/7 tests passed), OWASP ASVS v4.0 compliance mapping (77% alignment at L2 with partial L3 coverage), structured threat modelling across eight adversary classes, and a dynamic network security analysis conducted on 7 April 2026 using on-the-wire packet capture correlated with device-level runtime telemetry. The dynamic test provided the strongest empirical evidence for Ember's core claims: seven Diffie–Hellman ratchet steps were observed, each strictly triggered by a direction reversal, with the previous-chain-length field propagated and reset correctly at every transition; the verify-then-decrypt processing order was upheld for every inbound message; and the Yggdrasil link layer was shown to encapsulate overlay addressing such that a relay-adjacent observer sees only a uniform, fixed-size, fixed-cadence traffic profile structurally decoupled from message activity, with an exhaustive wire scan recovering none of the conversation identifier, device names, message identifiers, envelope structure, or overlay addresses.

The limitations of the current system are explicitly acknowledged and precisely bounded. Most importantly, the forward-secrecy guarantee is qualified by an identified implementation gap: the TTL cleaner deletes expired ciphertext but does not yet purge the corresponding per-message keys from the `MessageKeyStore`, leaving a residual recovery surface against a forensic adversary until `MessageKeyStore.purge(messageId)` is invoked atomically with ciphertext deletion. The X3DH handshake, while implemented, was not exercised end-to-end during the capture window and so remains a validation gap. Ember does not support group messaging, multi-device operation, or asynchronous offline delivery; it does not claim metadata anonymity against adversaries stronger than the relay-adjacent observer characterised here; and it does not guarantee physical secure deletion on Android storage media. Input validation coverage, side-channel resilience, and adversarial protocol robustness have not been empirically tested. These limitations are documented in the threat model and carried transparently through the evaluation, ensuring that Ember's claims are commensurate with its evidence.

Despite these limitations, Ember occupies a defensible position in the secure messaging design space. It demonstrates that the elimination of centralised infrastructure does not preclude strong content confidentiality, authenticated message integrity, per-message forward secrecy, or aggressive data minimisation. By avoiding third-party push notification services, central metadata aggregation points, pre-key servers, and server-mediated key management, Ember removes well-documented classes of privacy leakage that persist in mainstream encrypted messaging systems. The trade-offs it accepts (synchronous availability, foreground service resource costs, and a more intricate cryptographic state machine) are explicit, bounded, and appropriate for a research prototype that prioritises correctness and architectural clarity over feature completeness.

The most impactful directions for future work are now to *complete* rather than introduce the ratcheting story: closing the `MessageKeyStore` retention gap to deliver forward secrecy against a forensic adversary, validating the X3DH handshake end-to-end within a capture window, and subjecting the composed X3DH-plus-ratchet protocol to formal symbolic analysis and adversarial network testing. Each of these directions builds directly on the architectural foundation established by the current system and would extend Ember's guarantees into domains that are currently treated as validation gaps or explicit non-goals.

10 Conclusion

This paper has presented Ember, a serverless peer-to-peer messaging system that provides end-to-end encrypted communication with per-message forward secrecy and post-compromise security over a decentralised IPv6 mesh network. The system was designed to investigate whether secure messaging can be achieved without centralised infrastructure, under conditions where regulatory and architectural pressures increasingly threaten the viability of conventional end-to-end encryption deployments. Rather than proposing a novel cryptographic primitive, the work focused on system-level composition: integrating

an Extended Triple Diffie–Hellman (X3DH) key agreement and a Double Ratchet key-evolution protocol into a coherent, analysable, and practically deployable architecture on mobile platforms, without the coordinating infrastructure on which Signal-style deployments ordinarily rely.

Ember makes five concrete, implemented contributions. First, it demonstrates that serverless, peer-to-peer messaging over an encrypted overlay network is technically viable on Android, with steady-state end-to-end delivery latencies around 202 ms (median) and a per-direction-reversal Double Ratchet step costing only approximately 17 ms, which is negligible relative to the dominant per-connection TCP establishment cost. Second, it integrates X3DH initial key agreement with Double Ratchet per-message key evolution to provide forward secrecy and post-compromise security on a per-message basis, with no central key-distribution service; the dynamic evaluation confirms that the ratchet genuinely ratchets, performing a DH step on every direction reversal with correct chain-skipping propagation. Third, it enforces a ciphertext-only persistence model in which plaintext message content is never written to disk, materially reducing the forensic value of stored application state. Fourth, it implements TTL-driven message expiration that deletes expired ciphertext records within approximately 100 ms of their nominal expiry, providing tight, predictable application-level ephemerality. Fifth, it adopts a layered architecture with explicit trust boundaries that separates UI logic, cryptographic operations, persistence, and network transport, making security-critical paths independently reviewable and testable.

Ember is offered not as a replacement for mature, widely deployed secure messaging platforms, but as a rigorous exploration of an alternative point in the design space. That could be one platform in which no central authority need be trusted, no persistent infrastructure need be maintained, and no third-party service need be consulted for communication to occur. Within the boundaries of its stated threat model, the system achieves what it set out to achieve: serverless, peer-to-peer, end-to-end encrypted messaging with per-message forward secrecy, minimal retained state, empirically validated ratcheting behaviour, and security properties that are explicit, bounded, and defensible.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Hamish Alsop and Leandros Maglaras; methodology, Hamish Alsop; software, Hamish Alsop; validation, Hamish Alsop and Naghmeh Moradpoor; formal analysis, Hamish Alsop; investigation, Hamish Alsop and Leandros Maglaras; resources, Hamish Alsop; data curation, Hamish Alsop and Naghmeh Moradpoor; writing—original draft preparation, Hamish Alsop; writing—review and editing, Hamish Alsop, Leandros Maglaras and Naghmeh Moradpoor; visualization, Hamish Alsop; supervision, Leandros Maglaras; project administration, Hamish Alsop. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the Corresponding Author, Hamish Alsop, upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Akalin N, Giaretta A. From chat control to robot control: implications of the chat control proposal for human-robot interaction. arXiv:2601.02205. 2026.
2. Abelson H, Anderson R, Bellovin SM, Benaloh J, Blaze M, Callas J, et al. Bugs in our pockets: the risks of client-side scanning. *J Cybersecur*. 2024;10(1):tyad020.
3. Unger N, Dechand S, Bonneau J, Fahl S, Perl H, Goldberg I, et al. SoK: secure messaging. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*; 2015 May 17–21; San Jose, CA, USA. p. 232–49.
4. Onik AR, Brown J, Walker C, Baggili I. A systematic literature review of secure instant messaging applications from a digital forensics perspective. *ACM Comput Surv*. 2025;57(9):1–36. doi:10.1145/3727641.
5. Collins D, Riepel D, Tran SAO. On the tight security of the double ratchet. In: *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*; 2024 Oct 14–18; Salt Lake City, UT, USA. p. 4747–61.
6. Li Z, Xiao M, Xu R. Formal analysis of signal protocol based on logic of events theory. *Sci Rep*. 2024;14(1):20606. doi:10.1038/s41598-024-71666-y.
7. Xiao M, Wan H, Fan H, Shao H, Li Z, Yang K. Formal analysis of ratchet protocols based on logic of events. *Appl Sci*. 2025;15(13):6964. doi:10.1038/s41598-024-71666-y.
8. Hashimoto K, Katsumata S, Kwiatkowski K, Prest T. An efficient and generic construction for signal's handshake (X3DH): post-quantum, state leakage secure, and deniable. *J Cryptol*. 2022;35(3):17. doi:10.1007/s00145-022-09427-1.
9. Unger N, Goldberg I. Improved strongly deniable authenticated key exchanges for secure messaging. *Proc Priv Enhancing Technol*. 2018;2018(1):21–66. doi:10.1515/popets-2018-0003.
10. Barnes R, Beurdouche B, Robert R, Millican J, Omara E, Cohn-Gordon K. The messaging layer security (MLS) protocol. RFC 9420, IETF. 2023 [cited 2026 Jan 1]. Available from: <https://datatracker.ietf.org/doc/rfc9420/>.
11. Wallez T, Protzenko J, Beurdouche B, Bhargavan K. TreeSync: authenticated group management for messaging layer security. In: *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*; 2023 Aug 9–11; Anaheim, CA, USA. p. 1217–33.
12. Kirsh LB. Message-layer encryption in Ricochet. 2017 [cited 2026 Jan 1]. Available from: <https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1141>.
13. Lewis SJ. Cwtch: privacy preserving infrastructure for asynchronous, decentralized, multi-party and metadata resistant applications. Vancouver, BC, Canada: Open Privacy Research Society; 2018.
14. Blöchinger S, von Seck R. Survey of mesh networking messengers. *Network*. 2021;1:163–81.
15. Song Y. Cryptography in the wild: an analysis of briar. Zürich, Switzerland: ETH Zürich; 2022.
16. Floury T. Experimenting with matrix federation over Yggdrasil. 2019 [cited 2026 Jan 1]. Available from: <https://www.epfl.ch/labs/dedis/wp-content/uploads/2020/01/presentation-2019-2-Timothee-Floure-Experimenting-with-Matrix-federation-over-Yggdrasil.pdf>.
17. Pestov O, Kyrychek H. Yggdrasil routing scheme as a basis for large-scale decentralized networking. In: *Proceedings of the CEUR Workshop Proceedings*; 2024 Sep 23–25; Odesa, Ukraine.
18. Vidas T, Votipka D, Christin N. All your droid are belong to us: a survey of current android attacks. In: *Proceedings of the 5th USENIX Conference on Offensive Technologies*; 2011 Aug 8; San Francisco, CA, USA.
19. Reardon J, Capkun S, Basin D. Secure deletion on log-structured file systems. *ACM Trans Storage*. 2016;12(2):1–32. doi:10.1145/2414456.2414493.
20. Anglano C. Forensic analysis of WhatsApp messenger on Android smartphones. *Digit Investig*. 2014;11(3):201–13. doi:10.1016/j.diin.2014.04.003.
21. Heath H, MacDermott Á, Akinbi A. Forensic analysis of ephemeral messaging applications: disappearing messages or evidential data? *Forensic Sci Int Digit Investig*. 2023;46:301585.
22. Schnitzler T, Utz C, Farke FM, Pöpper C, Dürmuth M. Exploring user perceptions of deletion in mobile instant messaging applications. *J Cybersecur*. 2020;6(1):tyz016. doi:10.1093/cybsec/tyz016.
23. Samarin N, Sanchez A, Chung T, Dan A, Juleemun B, Gilsenan C, et al. The medium is the message: how secure messaging apps leak sensitive data to push notification services. *Proc Priv Enhancing Technol*. 2024;2024(4):967–82. doi:10.4324/9781315080918-31.

24. Blessing J, Anderson RJ, Beresford AR. KeyDroid: a large-scale analysis of secure key storage in Android apps. arXiv:2507.07927. 2025.
25. Nelson B, Pagnin E, Askarov A. Metadata privacy beyond tunneling for instant messaging. In: Proceedings of the 2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P); 2024 Jul 8–12; Vienna, Austria. p. 697–723.
26. Kuhn C, Beck M, Schiffner S, Jorswieck E, Strufe T. On privacy notions in anonymous communication. Proc Priv Enhancing Technol. 2019;2019(2):105–25. doi:10.2478/popets-2019-0022.