Breaking Cryptography in the Wild: The Loose Ends of the Wire

Master Thesis

Author(s):
Tsouloupas, Andreas

Publication date:
2023

Permanent link:
https://doi.org/10.3929/ethz-b-000673362

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Breaking Cryptography in the Wild: The Loose Ends of the Wire

Master Thesis

Andreas Tsouloupas

September 25, 2023

Advisors: Prof. Dr. Kenny Paterson, Matteo Scarlata, Kien Tuong Truong

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

In this thesis, we analyse the Wire secure messenger, focusing on the security of the end-to-end (E2E) protocols it implements. Wire claims that their messenger is used by governmental bodies of five of the seven G7 member countries, and the Wire app counts millions of downloads in mobile app stores. This makes it a valuable target for attackers.

To the best of our knowledge, Wire's cryptographic design has not undergone an extensive analysis, aside from an independent two-phase audit that primarily emphasized secure programming practices.

Wire currently supports two E2E protocols: Proteus, based on Signal's Double Ratchet protocol and which is used by default, and MLS, the newly standardized IETF "Messaging Layer Security protocol", which is still in active development.

We show that a combination of protocol-level and application-level vulnerabilities significantly undermine the confidentiality and authentication guarantees of Proteus E2E channels. These vulnerabilities allow a malicious server to tamper with message order, redirect messages to unintended groups, compromise group confidentiality, and even undermine Forward Secrecy (FS).

Furthemore, we study Post-Compromise Security (PCS) of Proteus, and find that Wire does not achieve PCS against strong adversaries with access to the private keys of the compromised users, nor against weaker adversaries with temporary oracle access to these keys.

Finally, we take a look at Wire's MLS integration, and find several early design flaws which negate many of the security benefits of MLS, leading to trivial confidentiality violations and not achieving PCS in the presence of a malicious server.

To complement our findings, we provide mitigations and recommendations aimed at enhancing the overall security of the messaging system subject of this study.

Contents

Chapter 1

Introduction

Instant messaging plays a key role in facilitating daily communication for billions of individuals across the globe. There are hundreds of messaging platforms/systems providing instant messaging services. Among these, some have achieved widespread popularity on the international stage. Examples include WhatsApp, Facebook Messenger, and Telegram. In parallel, there are messaging platforms whose prominence is mostly limited to specific nations, like WeChat and QQ in China [36]. Collectively, these messaging platforms serve billions of users [52] on a daily basis.

Messaging platforms typically adopt an architecture that involves devices, commonly referred to as *clients*, engaging in communication with one another via a centralized server and over a network that inherently is untrusted: the Internet. The server plays an important role in routing messages to their intended recipients: the server has the potential to perform Denial-of-Service (DoS) attacks by simply refusing to relay messages to devices. In such cases, there are no measures available to prevent the server from carrying out this form of attack.

Additionally, messages traversing through the server may contain sensitive information: users of these platforms range from individuals exchanging their private information to large-scale global organizations using them for business-critical discussions. Therefore, securing the confidentiality and authenticity of the information transmitted through the server is of significant importance. As an initial layer of security, client-to-server (C2S) communication is commonly protected against network-based attacks using protocols like TLS and QUIC. These protocols serve to hide the content and metadata of messages from network adversaries but, crucially, not against a malicious or compromised server.

Secure Messaging In messaging platforms that solely secure C2S communications, a crucial component of the system's architecture, the server, is

given ultimate trust. Such messaging servers could misbehave, or be a tempting target for potential external attackers due to its inherent value, and it can be exploited for unethical purposes like censorship, surveillance, and targeted advertisement. Consequently, in secure messaging, it becomes important to view the intermediate server as a potential malicious actor or a compromised component operating with malicious intent. This realization underscores the necessity for the existence of end-to-end (E2E) channels. Essentially, the objective is to secure the communication between two devices from device to device, ensuring that messages exchanged between them, even if they are routed through an untrusted server, remain confidential and authentic. Specifically, the basic security properties expected from E2E channels are *confidentiality*, *integrity*, and *authenticity*. In other words, neither the server nor any other entity, except the endpoint devices, can read, modify, or insert messages into an E2E channel between the devices.

While confidentiality, integrity, and authenticity might initially appear to be an adequate set of prerequisites for E2E channels, the reality is more complex. Unlike other short-lived E2E channels established through protocols like TLS, the E2E channels in a secure messaging system persist for extended periods, possibly spanning many years. This extended lifespan introduces a realistic threat: an attacker gaining access to a device and corrupting all the secrets currently in use for E2E communication. When combined with the capability to store all previously exchanged messages, possibly due to the attacker's control over the server responsible for forwarding encrypted messages, the consequences can be disastrous. In the scenario where all past messages are encrypted with keys that the adversary can derive with the corrupted secrets, the confidentiality of the entire communication, even before the corruption occurred, is compromised. Furthermore, if the secrets are never updated, the attacker can exploit the E2E channel to transmit encrypted messages using the compromised secrets, thereby undermining the integrity and authenticity of future messages as well as their confidentiality. Clearly, the need arises for advanced security properties that can guarantee protection both before and after a corruption. These properties are *Forward Secrecy* (FS) and *Post-Compromise Security* (PCS).

Forward Secrecy [24] is concerned with the confidentiality of messages sent before the corruption occurs. On the other hand, Post-Compromise Security [26] is concerned with the confidentiality, integrity, and authenticity of messages exchanged after the corruption, which implies that the E2E channel can recover the above mentioned security properties under certain circumstances. Last, a mechanism enabling users to prevent (or detect) a malicious server from impersonating others has to be provided. In this work, a messenger is considered secure if and only if it can satisfy the above requirements and properties, which is reasonable for our target messenger's high-risk users.

Many modern secure messaging applications, such as WhatsApp, Signal, and Facebook Secret Conversations, use Signal's protocol and its open-source implementation libsignal [41]. This protocol is based on the *Double Ratchet* algorithm, which offers the properties necessary for secure messaging. WhatsApp is currently the most popular messenger, with over 2 billion monthly active users [52], and by default provides end-to-end encryption (E2EE) for all communications.

Group Messaging Typically, users communicate with one or more other users in conversations, possibly using multiple devices. Group messaging allows the devices of all users in a conversation to simultaneously communicate in groups of many devices rather than just a single device. In the context of the Double Ratchet algorithm, which many E2E channels use for the communication between two devices, group messaging is realized by combining pairwise channels among all N devices that take part in a group; hence, each message in a group requires $N - 1$ encryptions: one for each pairwise channel.

Recently, the Internet Engineering Task Force (IETF) standardized a new protocol for E2E communication known as the Messaging Layer Security (MLS) [13]. MLS targets efficient E2E encrypted group messaging, and has been designed to provide Forward Secrecy and Post-Compromise Security. It is widely anticipated that many instant messaging service providers will adopt MLS in the future: the European Commission's Digital Marketing Act (DMA) [27] mandates that all major messaging platforms become interoperable starting in 2024, and MLS is set to play a central role in achieving secure E2E communication interoperability, with industry giant Google already signaling its commitment by announcing plans to integrate MLS into its communications services [35].

The Target In this thesis, we study the E2E communication protocols employed in Wire. Wire, founded in 2012 by some of the original Skype founders, offers its services to individuals, businesses, and governmental organizations. Significantly, as claimed by Wire [33], their products are adopted by organizations entrusted with mission-critical information, including large corporations and governmental bodies, even extending to five of the world's G7 governments. Additionally Wire's application in Google's play store counts over one million downloads [3]. This aspect makes the examination of Wire's E2E communication particularly interesting, given that any flaw discovered could have severe impact.

To this day, Proteus, Wire's bespoke protocol utilized for E2E communication, is the main protocol Wire uses in E2E channels to enable secure messaging. Proteus implements a variant of Signal's protocol and Signal's

Sesame [44] session management. In our study, we conduct a comprehensive examination of Proteus and reveal the security implications of its minor deviations from the Signal protocol. Additionally, we place a significant focus on session management and explore its implications on PCS.

Wire was also involved in the development of the MLS standard and they aim to be the first to deliver the benefits of MLS to their end users, with ongoing efforts to implement a stable MLS support in Wire's applications. MLS in Wire is an interesting area of study, particularly with regards to the decisions made by Wire concerning choices that are left to the application developer within the MLS standard. Wire indicates that Proteus is suitable for one-to-one and small groups, while MLS is better for large groups [31]: both protocols may coexist in Wire in the future, meaning that users' conversations will be using either Proteus or MLS as the underlying protocol.

1.1 Communication Abstraction

A *user*, typically a human, engages in a messaging system, often using multiple *devices* like smartphones and laptops. Each device is equipped with a platform-specific application that creates *clients* and where they reside with cryptographic keys for E2E communication.

When studying a messaging system, we distinguish different levels of abstraction at which communication happens. For instance, when two or more users have a *conversation*, the messaging clients in their devices maintain *dialogues*, which might be formed of many low-level *session* of a messaging protocol, within a certain *group*. Below we list the different terms we use to refer to communication at different abstraction levels, and give detailed description of what they represent.

Conversation A *conversation* is the high-level communication between different users, regardless of the number of users, how their clients exchange the actual messages and how the multiple clients of a user synchronize between each other. More concretely, from the point of view of a user, a conversation is represented by the chat visible on their device: the chat should be consistent across the different devices of the user, and show messages as originating from other users rather than from particular devices.

Group A *group* is the abstract many-to-many communication between clients. We refer to each client as a *group member*. Group communication can be obtained by combining pairwise dialogues between members of the group. Every conversation is connected to a group, meaning that when users send a message in a conversation using one of their clients, the client sends a

message to the underlying group, which utilizes the pairwise dialogues to communicate with the other group members.

Dialogue A *dialogue* is an abstract E2E (1-to-1) channel between two clients. Clients use a dialogue with another client to communicate, while the internal details of the dialogue are opaque to them. A dialogue might be implemented as a single session or a *session management protocol*. The latter is a protocol that uses multiple sessions to communicate with details hidden to the clients. In Signal, the Sesame protocol is used as a dialogue between clients, while in this work, we present Proteus dialogues.

Session A *session* is the low-level concrete E2E channel between two clients, and it is a single instance of the communication protocol. Clients employ dialogues for E2E communication, which use sessions as a building block. In Signal and Wire, (slightly different) Double Ratchet-based protocols realize sessions.

1.2 Related Work

Many studies in recent years have focused on the security of messaging platforms used by millions of people around the globe. More specifically, of high interest are applications explicitly claiming strong security guarantees, such as Telegram, Matrix, Threema, Signal and our target Wire.

Security Analysis of Wire As far as we know, there has been no extensive analysis of Wire's cryptographic design. Wire released a high-level description of their cryptographic components, various procedures for user and client registration, and authentication in [34]. Furthermore, there is a two-phase audit review of Wire [49, 48], mainly focusing on secure programming rather than the cryptographic components and their usage in Wire's applications. The first phase [49] concentrates on the cryptography libraries codebase, while the second phase [48] focuses on Wire's applications. These audits failed to uncover vulnerabilities that are responsible for the attacks detailed in Chapter 4. The first phase of the audit revealed a vulnerability enabling the use of degenerate (small order) Curve25519 points in Diffie-Hellman computations, resulting in predictable derived shares. This vulnerability had the potential to lead to DoS attacks by exhausting all available ephemeral values that clients upload to the server. This issue was addressed starting from Proteus v0.6.0.

Security Analysis of Other "Secure" Messengers In [38], Jakobsen and Orlandi show that an older version of the symmetric encryption scheme

used in Telegram (MTProto 1.0 [54]) was not IND-CCA secure. A subsequent study conducted by Albrecht et al. in [5] investigates the security of Telegram and, in particular, the updated version of their symmetric encryption scheme (MTProto 2.0 [53]). Despite the existence of standard and well-studied cryptographic primitives, Telegram differentiates MTProto 2.0 from other well-studied schemes by using unconventional constructions. For instance, Telegram employs bespoke MAC and KDF schemes; they use an Encrypt-and-Mac construction, and encryption operates in Infinite Garble Extension (IGE) mode, which is not commonly used. In addition, client-to-server and server-to-client keys are obtained from a raw Diffie-Hellman shared secret and have overlapping bits. Several attacks were discovered due to the protocol's eccentric nature, some of which are more theoretical than practical; however, this still indicates the fragility of a highly customized protocol.

Albrecht, Celi, Dowling and Jones study Matrix and its Olm and Megolm channels for secure messaging in [4]. The paper's authors found several practical attacks, most of them requiring the collaboration of an active malicious homeserver. In Matrix, a homeserver is the intermediate server used to forward the end-to-end encrypted messages from a user's device to the other room users' devices, where a room is the equivalent of a conversation between one or more users. The lack of encryption and authentication of room management messages enabled a malicious homeserver to trivially violate the confidentiality of messages by adding new adversarially-controlled users to a room or devices to existing users. Moreover, in another attack, the malicious homeserver and an insecure implementation choice permitted by the specification allowed an adversary to impersonate a target device. The impersonation attack comes in two versions, the semi-trusted and trusted, where the latter builds on the former to escalate the trust level of the impersonation. The attacks above illustrate the importance of ensuring that an untrusted component (according to our threat model) involved in our protocol cannot actively or passively violate the security properties we claim.

In [47], Paterson, Scarlata and Truong investigate Threema, another messenger claiming strong security guarantees. Once more, a messaging application yielded disappointing results. The authors presented many attacks, including one that exploits the lack of integrity protection on metadata sent with encrypted messages, allowing for manipulation by a malicious server.

As exploitable cryptography-related vulnerabilities are continually discovered, we anticipate the emergence of many more in the future. In this work, we present attacks inspired from and exploiting similar vulnerabilities found in previous studies.

Formalization Amongst the security properties of messengers offering end-to-end encryption are FS and PCS. FS guarantees that messages already sent in an E2E channel remain secure even in the presence of an adversary who learns all the secrets and private long-term keys currently possessed by a communicating party. Concretely, having forward secrecy means that an adversary cannot run a "store now, decrypt later" attack: the long-term secrets do not help the attacker in decrypting old messages.

On the other hand, PCS (formalized in [26]) is concerned with the security of the E2E channel after compromise. In particular, integrity, authenticity and confidentiality are the most relevant security properties. In order to recover security, an update event should occur where the communicating parties securely exchange fresh secrets (e.g., using Diffie-Hellman key exchange) and combine them with compromised secrets in a forward secure way, possibly by using a one-way function. If the adversary is passive when the parties exchange fresh secrets, meaning the adversary does not have access to the latest secrets, then the new entropy introduced "heals" the E2E channel.

Cohn-Gordon et al. perform a formal security analysis of Signal in [25]. The Signal messaging protocol consists of the extended triple Diffie-Hellman (X3DH) for the initial key agreement and the Double Ratchet algorithm. As opposed to the work conducted by Fischlin and Günther in [32] which introduced for the first time a multi-stage key exchange model to address the limitations of the single-stage key exchange models, Cohn-Gordon et al. define a novel multi-stage key exchange model (highly inspired by the one in [32]), which allows for a tree of stages rather than just a sequence.

PCS Experiment An experiment conducted by Cremers et al. in [28] gave some very interesting results about the PCS of various messengers against a so called Clone Attack where the full state of a user's device is compromised by an attacker who clones the device. The expectation is that, after the original device exchanges a few messages with other peers, their E2E channels heal according to the PCS guarantees of the underlying communication protocols. The experiment grouped the messaging applications into three classes. The first category contains applications using the Signal Protocol and its open-source implementation [41], such as Signal, WhatsApp and Facebook's Secret Conversations (according to their respective security whitepapers [51, 56, 45]). The second category considers applications based on Signal's protocol; however, they implement it from scratch with possibly some variations. Some examples are Viber [55] and Wire [34]. Last, applications using protocols other than the Signal protocol or some variant are included in the third class. In this class, we can find Threema and Telegram. As per the experiment's configuration, the clone device should not be able to decrypt certain messages exchanged between the original devices after cloning, violating PCS. Wire had the strangest and most difficult to explain

behavior among the messengers using a protocol based on Double Ratchet. More specifically, the Wire clone client successfully decrypted two messages from the counterparty sent in the test phase, while the expectation was at most one message. In Section 4.4.6, we delve into the underlying causes of the unusual behavior observed in Wire.

Although many messaging applications use the same Double Ratchet protocol used in the Signal application, with some of them even sharing code with Signal's, the behavior seems to diverge as a result of some implementation choices.

Session Management In a study by Cremers, Jacomme, and Naska [30], which is independent of our work, they investigate Sesame [44], Signal's session management protocol. Before their study, security properties like FS and PCS were exclusively examined at the per-session level, with Signal's eXtended Triple Diffie-Hellman and Double Ratchet protocols being notable for formally proven strong FS and PCS guarantees.

However, Cremers et al. demonstrates that when multiple sessions are employed within a dialogue between two communicating clients, the way in which they are handled becomes pivotal and can undermine PCS. Indeed, they experimentally show that Sesame falls short of guaranteeing PCS, assuming an adversary with the capability to clone a device's complete state, and thereby gaining implicit access to its long-term identity key pair. Considering PCS at the dialogue level used by clients at the application level, which utilizes a session management protocol rather than directly a session, is necessary: a user of the application does not see PCS at the session level, but can only interact with the conversation's application interface.

Their discoveries share commonalities with our own findings regarding the way Wire handles sessions, though their results have been discovered independently. Wire's session management protocol is referred to as Proteus dialogues, in this work.

While Wire's session management takes inspiration from Sesame, it has significant differences setting it apart. The differences, as we will see in our analysis, require more changes to mitigate Proteus dialogues' weaknesses than the ones proposed in [30].

In our work, we extend the analysis to consider PCS from the standpoint of a strong adversary capable of fully compromising a device's long-term identity key pair. Additionally, we explain that discontinuing session management is a requisite step to achieve PCS under this more stronger adversarial setting.

MLS The IETF Messaging Layer Security (MLS) working group recently standardized MLS (under RFC 9420 [13]), a new asynchronous group messaging protocol based on tree structures. MLS attempts to efficiently and continuously derive group secrets while simultaneously providing advanced security properties such as FS and PCS. During the development process of MLS, many academic works analyzed the security guarantees that various drafts or components were providing [7, 22, 6, 11, 9, 10, 23, 8, 29]. Many of them presented practical and theoretical attacks as well as solutions that future drafts gradually adopted until reaching an MLS stable version for standardization.

Wire played a role in the standardization of MLS and are actively working to integrate MLS groups into their messaging platform. In this work, we analyze the specific choices made by Wire concerning open-to-the-application designer aspects within the MLS standard.

1.3 Contribution and Outline

Our main contributions are the following:

- We uncover the impact of session management using Proteus dialogues for E2E communication. More specifically, we present three attacks that defeat PCS guarantees at the application level despite the usage of the Double Ratchet algorithm as a building block of the underlying sessions. Additionally, we explain why E2E channels using session handling cannot achieve the same PCS guarantees as the ones provided by a single session.
- We propose a fix that improves PCS for Wire's session management using improved Proteus dialogues. Additionally, we suggest an alternative dialogue solution that achieves stronger PCS, which utilizes a single session and a tiebreaker rule for resolving simultaneous session initializations.
- We show that Wire does not enforce message ordering nor display of messages within the intended Proteus group. A malicious server is able to reorder messages and redirect messages to different Proteus groups due to the lack of integrity protection of the fields carrying this information.
- We highlight the inadequacy of the current client registration and multi-device support in Wire, which essentially allows a malicious server to add a new malicious client to any user's client list, resulting in a trivial confidentiality violation.
- We show how a signature omission in information uploaded to the server by the clients in order to enable Proteus E2E communication

degrades FS.

- We discuss decisions made by Wire regarding MLS and their security implications, especially when clients are removed from MLS groups and for the root of trust for clients' credentials. We explain why removing a member from a group does not take immediate effect and how Wire can trivially impersonate any client.

Chapter 2 provides the preliminaries for this work. Then, in Chapter 3, we present an overview of the Wire messenger including both Proteus and MLS E2E solutions. The Proteus E2E communication protocol is described in more detail as it is implemented by Wire. For MLS, we only provide a high-level description of its fundamental operations as defined in its standard and we highlight important Wire-specific choices. For each of the attacks and findings which resulted from our analysis of Proteus and MLS (as discussed in Chapters 4 and 5, respectively), we describe mitigations that improve the overall security of the messaging system. We conclude in Chapter 6 by summarizing our results and providing guidance for potential future work.

Chapter 2

Preliminaries

This chapter begins with the notation used throughout the thesis for describing protocols and procedures (Section 2.1). Then, in Section 2.2, we provide a high-level discussion of the cryptographic primitives used in Wire's Proteus protocol. Subsequently, in Section 2.3, we describe in more detail Forward Secrecy and Post-Compromise Security, the expected (in this work) advanced security properties of a secure messenger. Finally, Section 2.4 provides a high-level explanation of the Double Ratchet algorithm. We refer to Section 3.7 and [13] for an introduction to MLS

2.1 Notation

We use standard notation to describe protocols and procedures:

- $\{0,1\}^n$: the set of bit strings of length n.
- $a \| b$: the concatenation of the big-endian representation of the integer values a and b, in that order.
- $m_1 \| m_2$: the concatenation of the byte string m_1 and m_2, in that order.
- $a \oplus b$: bitwise XOR of the two bit strings.
- $a \leftarrow b$: assign the value of b to a.
- $a \leftarrow\!\!\$\ \mathcal{S}$: sample an element from finite set $\mathcal{S}$ uniformly at random and assign its value to a.
- $a \leftarrow \mathsf{F}()$: assign the output of the deterministic function F to a.
- $a \leftarrow\!\!\$\ \mathsf{F}()$: assign the output of the probabilistic function F to a.

Let G represent a group, and let g serve as its generator. Throughout this thesis, we will employ multiplicative notation for group operations, despite

the operations being conducted on an elliptic curve where additive notation might be more fitting. We choose this approach because multiplicative notation is more widely used in the literature. For instance, consider two private keys x and y, then their corresponding public keys are $X = g^x$ and $Y = g^y$, respectively. Additionally, we omit the details of (elliptic curve) Diffie-Hellman ((EC)DH) computation and we denote the derived secret by $g^{x \cdot y} = X^y = Y^x$.

2.2 Cryptographic Primitives

Wire reuses several standard cryptographic primitives in order to build its bespoke Proteus E2E protocol. In the following Sections, we provide high-level discussions and interfaces for the encryption algorithm (Section 2.2.1), the message authentication code (Section 2.2.2), the elliptic curve employed for Diffie-Hellman computations (Section 2.2.3), and the key derivation function (Section 2.2.4) used as underlying building blocks of Proteus.

2.2.1 ChaCha20 Encryption Algorithm

Encryption algorithms are responsible for providing confidentiality to messages at rest and exchanged over untrusted networks. ChaCha20 is a stream cipher which Bernstein describes in [20]. It is an improvement in terms of efficiency and cryptanalysis of the Salsa20 stream cipher, also described by Bernstein in [19]. Additionally, it is considerably faster (around three times) than AES on non-specialized hardware.

The ChaCha20 encryption algorithm repeatedly calls the ChaCha20 block function, with the same key and nonce, and a counter that is incremented on each call by one. The successive calls of the block function generate a sufficiently long keystream, which is then XOR-ed with the given plaintext or ciphertext. The ChaCha20 encryption algorithm is described in [46] Section 2.4. The encryption algorithm provides the following interface:

- $c \leftarrow \mathsf{ChaCha20.Enc}(k, n, m)$: given a key $k \in \{0,1\}^{256}$, a nonce $n \in \{0,1\}^{96}$, and an arbitrary length plaintext m, output a ciphertext c. The starting counter used in the sequential calls of the ChaCha20 block is set to a constant value by the library Wire utilizes.
- $m \leftarrow \mathsf{ChaCha20.Dec}(k, n, c)$: given a key $k \in \{0,1\}^{256}$, a nonce $n \in \{0,1\}^{96}$, and an arbitrary length ciphertext c, output a plaintext m. The starting counter used in the sequential calls of the ChaCha20 block is set to a constant value by the library Wire utilizes.

Many papers [12, 37, 50] discuss the security of the ChaCha and Salsa families of stream ciphers. Notably, ChaCha20 has an unfortunate property which it shares with other stream ciphers: reusing the same nonce n (and

counter) with the same key k results in the same keystream, which then implies an attack if two different ciphertexts c_1 and c_2 are encrypted using the same keystream, with $c_1 \oplus c_2 = m_1 \oplus m_2$. Hence, nonces must be chosen carefully, typically using sequence numbers incremented by one for each encryption/decryption; otherwise, repetition can be exploited with cryptanalysis or partial knowledge of one of the plaintexts.

2.2.2 HMAC-SHA256 Authenticator

A Message Authentication Code (MAC) scheme offers a method for verifying the integrity of data sent over or kept within an unreliable medium. Typically, a MAC scheme is used in a group of parties sharing a secret key to verify that a message was sent from a group member. Notice that MAC schemes do not provide non-repudiation, meaning that any member of the group could have sent the message.

Wire uses the HMAC construction, which Bellare, Canetti, and Krawczyk present and analyze in [17]. The HMAC construction uses a cryptographic hash function (in Wire SHA256) with a secret key in order to generate an authentication tag used for verification. The interface provided by HMAC instantiated using SHA256 is the following:

- $\tau \leftarrow \mathsf{HMAC}_{\mathsf{SHA256}}.\mathsf{Tag}(k, m)$: given a key k and a message m (size limited to the capabilities of the underlying hash function), output the MAC tag $\tau \in \{0,1\}^{256}$ for m.
- $v \leftarrow \mathsf{HMAC}_{\mathsf{SHA256}}.\mathsf{Vfy}(k, \tau, m)$: given a key k, a MAC tag τ, and a message m, output zero when verification fails; otherwise, one ($v \in \{0,1\}$).

We highlight that Wire uses the Encrypt-then-MAC composition of ChaCha20 and HMAC.

2.2.3 Curve25519

Curve25519 is a Montgomery elliptic curve over the prime field defined by the prime number $2^{255} - 19$. It was designed by Bernstein in [18] for efficient elliptic curve Diffie-Hellman (ECDH) key exchange. Both public and private keys are 32 bytes long because ECDH share computation operates only on the X-coordinate, which allows efficient use of the Montgomery ladder algorithm.

In [21], the twisted Edwards curve birationally equivalent to Curve25519 is recommended for use in the EdDSA signature scheme. In this particular choice of curve, the name Ed25519 is used for the signature scheme. On a high level, the interface provided for the Ed25519 signature scheme is the following:

- $(x, X) \leftarrow\$ \mathsf{Ed25519.KeyGen}()$: generates a private key x and the corresponding public key X. For simplicity, we use the generated public-private key pair without transformation for both ECDH computation and in the Ed25519 signature scheme.
- $\sigma \leftarrow \mathsf{Ed25519.Sign}(x, m)$: given a private key x and message m, output the signature σ for m.

2.2.4 HKDF-SHA256 Key Derivation Function

A Key Derivation Function (KDF) is an important cryptographic component that allows one to derive one or more cryptographically strong secret keys from some initial keying material, which might also be unsuitable for direct cryptographic use.

Wire uses HKDF, an HMAC-based Extract-then-Expand key derivation function. HKDF provides two functionalities. The first functionality takes an initial keying material of which not every bit might be distributed uniformly at random and extracts a fixed-length pseudorandom key from it. The extracted pseudorandom key concentrates the possibly spread entropy of the initial keying material, thus making the extracted key suitable for cryptographic use (whitening transformation).

The second functionality expands a pseudorandom key, typically derived from the extract functionality, to many other keys according to our needs and the cryptographic algorithms we use.

In [40], Krawczyk presents and analyzes HKDF. The interface of HKDF instantiated with HMAC-SHA256, as seen in Section 2.2.2, is the following:

- $k \leftarrow \mathsf{HKDF}_{\mathsf{SHA256}}.\mathsf{Extract}(s, k_{in})$: given a salt $s \in \{0,1\}^{256}$ and an initial keying material k_{in} of arbitrary length (limited by the capabilities of the underlying hash function), output a pseudorandom key $k \in \{0,1\}^{256}$.
- $k_{out} \leftarrow \mathsf{HKDF}_{\mathsf{SHA256}}.\mathsf{Expand}(k, \mathsf{info}, L)$: given a pseudorandom key $k \in \{0,1\}^{256}$, an arbitrary length label info (can be a zero-length string) and the output keying material length L in bytes ($\leq 255 \cdot 256$), output keying material k_{out} of L bytes.
- $k_{out} \leftarrow \mathsf{HKDF}_{\mathsf{SHA256}}(s, k_{in}, \mathsf{info}, L)$: equivalent to applying extract and then expand using the output of extract as the pseudorandom key k.

2.3 Secure Messaging Advanced Properties

The following discussion describes FS and PCS, two properties we expect from a secure messenger. Signal's Double Ratchet protocol achieves both of these properties in their strongest form [25]. We highlight that in the following discussions, we use the term E2E channel because it can be realized as

a dialogue with a single session or a collection of sessions using a session management protocol.

2.3.1 Forward Secrecy (FS)

FS is concerned with the security of communications that happened prior to any corruption. In particular, it ensures the confidentiality of previous communications between parties. This implies that, if an adversary compromises the current state of an E2E channel and has access to ciphertexts of previously encrypted messages, they cannot decrypt those ciphertexts by deriving their message keys.

More concretely, FS guarantees that an adversary who corrupts the complete state of an E2E channel, including the long-term key pairs, cannot compromise the security of messages exchanged within the E2E channel before the corruption. This implies that the long-term key pairs alone do not allow for the computation of any of the previous secrets used to derive previous message keys (e.g., the initial key exchange depends on a DH share computed using ephemeral key pairs). FS can be achieved by using a one-way function, such as a KDF, to ensure that the same message key is never used to encrypt more than one message, and knowledge of a message key does not allow for the derivation of past message keys.

2.3.2 Post-Compromise Security (PCS)

PCS is concerned with E2E channel security after a corruption has occurred. It ensures that the communicating parties' channel will regain the security properties of confidentiality, integrity, and authenticity. To achieve this, a channel state update is required to happen periodically. This event of updating the state of a channel is known as the healing step.

There are various forms of PCS that depend on the capabilities of the adversary. In this study, we consider PCS in the presence of either a weak or a strong adversary, referred to as PCS via weak or total compromise in [26], respectively.

In *weak-PCS*, the adversary gains temporary access to the long-term key pairs of the communicating parties through an oracle (along with the channel state). This scenario is particularly relevant when an adversary temporarily accesses a hardware security module (HSM) that stores long-term key pairs and can perform operations with the private keys without revealing them. Once the adversary loses access to the HSM, the parties can use their long-term key pairs to compute a token that the adversary cannot precompute when having access through the oracle to the private long-term keys (e.g., with a Key Encapsulation Mechanism (KEM)). Then the parties

can derive the new uncompromised secret using the corrupted secret of the previous stage and the token mentioned above as input to a KDF.

In *strong-PCS*, the adversary gets access to the actual long-term key pairs of the communicating parties. In this case, we require that the adversary remains passive during the healing step, and the channel only regains security after healing. Typically, this healing process involves asymmetric cryptography, with the introduction of ephemeral key pairs and the computation of a DH shared secret. Since the adversary is passive during the healing step, only the communicating parties possess knowledge of the DH share, effectively locking out the adversary. This derived share can then be used as input to a KDF (possibly alongside a previously compromised secret) to generate new uncompromised suitable for cryptographic usage secrets.

2.4 Double Ratchet

The Double Ratchet algorithm [42] is at the core of Signal's protocol and is responsible for encryption and decryption of messages within a session. It ensures that the established session provides both FS and strong-PCS through the use of two distinct ratchets: the asymmetric and the symmetric. A ratchet is a key schedule, where new secrets are continuously generated by KDF calls/steps. The asymmetric ratchet connects all symmetric ratchets together as we explain below.

A prerequisite of the Double Ratchet algorithm is a shared secret between the two communicating parties. Signal's protocol satisfies the requirement using the eXtended Triple Diffie-Hellman (X3DH) [43] authenticated key exchange algorithm. In the following discussions, we provide high level description of the asymmetric and symmetric steps.

Asymmetric Steps In a session, the two parties have an asymmetric ratchet and two types of symmetric ratchets: the receiving and the sending (symmetric) ratchets. The asymmetric ratchet begins with a root key derived from an initial key exchange (e.g. X3DH), and it strictly orders the symmetric ratchets one after the other, with each party assigned a sending ratchet exactly every two symmetric ratchets in the order specified by the asymmetric ratchet while the rest are receiving ratchets. Whenever a party wants to send a message after receiving messages on the latest symmetric ratchet, which is a receiving ratchet, then an asymmetric step forward is required so that it starts using its next sending ratchet. During the asymmetric step, the party who wants to send a message generates an ephemeral key pair. This ephemeral key pair, combined with the public ephemeral key of the peer from the previous asymmetric step, is used to compute a DH secret. The DH secret and a root key from the previous step are the inputs to a KDF to

derive the next root key and a chain key. The chain key is used in symmetric steps to derive message keys. The peer is informed about the asymmetric step through message metadata. The asymmetric steps provide both FS and strong-PCS because of the usage of a KDF with fresh ephemeral secrets and the previous root key.

Symmetric Steps At any given time, both parties hold chain keys for symmetric ratchets. A party uses a sending symmetric ratchet to send encrypted messages in the session, while it uses a receiving symmetric ratchet to receive encrypted messages. Whenever a party sends or receives a message on an existing symmetric ratchet, it performs a symmetric step. The symmetric step involves a KDF that takes the current chain key of the symmetric ratchet as input and outputs a new chain key and a message key for encryption or decryption. The symmetric steps provide only FS because of the usage of a KDF with the previous chain key but not an ephemeral secret.

Chapter 3

The Wire Messenger

This chapter presents Wire's usage scenarios and protocols relevant for our study, and it is followed by an analysis of findings in Chapters 4 and 5.

We begin with an overview of Wire's ecosystem, introducing the applications, servers, and entities involved in the typical use of the messaging system in Section 3.1. We then discuss Wire's selection of the client-to-server protocols in Section 3.2. Prior to introducing the end-to-end protocols, we explain the registration protocol in Section 3.3, which essentially enables multi-device support. In Section 3.4, we describe how Wire organizes cryptographic components and libraries across different levels of abstraction to support multiple end-to-end protocols and simplify compatibility across platforms. Proteus, Wire's own Double Ratchet protocol implementation, is extensively examined in Section 3.6. More recently, MLS has been added as a supported protocol together with Proteus. In Section 3.7, we briefly describe MLS, with detailed information available in the publicly accessible RFC [13].

3.1 Wire Ecosystem

Wire's ecosystem is depicted in Figure 3.1. A user is a participant in the messaging system. Users are denoted by A, B, U and V, each assigned a distinct *user identifier*. For a user U, $ID_U \in \{0,1\}^{128}$ denotes the user identifier given to the user, which is 16-bytes long. The presence of at least one client bound to a user is a prerequisite for a user to be able to exchange message in a conversation. In other words, a user can be envisaged as an entity possessing the knowledge of login credentials for an account and that is not bound to any cryptographic material. Further clarity on this distinction will emerge as we delve into the discussions on the C2S and registration protocols, addressed in Sections 3.2 and 3.3, respectively. Furthermore, a user is permitted to have a maximum of seven clients affiliated with them. The

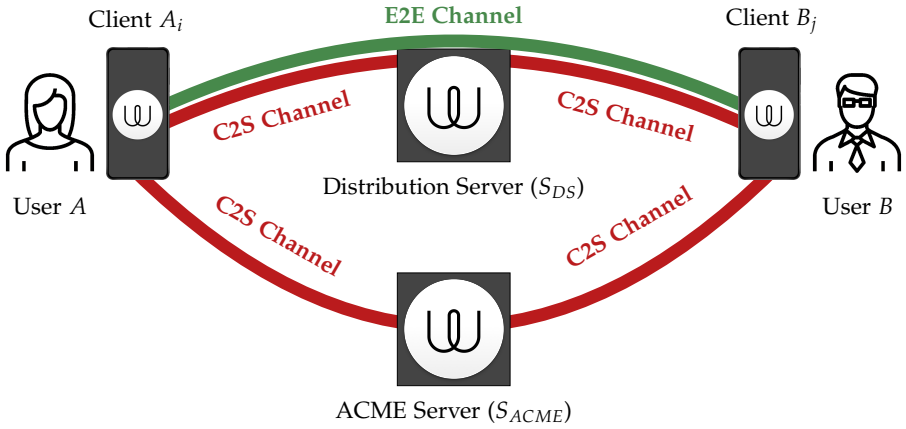


Figure 3.1: The different components and entities of the Wire service, highliting the different protocols used in Wire to provide discovery, certification and secure messaging between a user A with client A_i and B with client B_i.

clients of user U are denoted by U_i for $i \in \{0,1\}^{64}$, where i corresponds to the 8-bytes long *client identifier*. It is important that the client identifier remains distinct from those attributed to the user's other clients, so that each client is uniquely identified. Moreover, a client is bound to a single user. Functionally, a client is situated within a device, i.e., a smartphone or a laptop, that has installed one of the Wire applications; depending on the operating system and architecture of the device, different versions of the applications are available to the user. Unlike other secure messengers, in Wire's ecosystem, none of a user's clients can be seen as the primary client that brings the user's clients together, e.g. Signal has a concept of a primary client and secondary clients which a user can register with the involvement of the main client.

As previously discussed, the Wire ecosystem consists of different applications tailored for distinct platforms, encompassing iOS, web, desktop, and two Android applications: the legacy and the new. When conducting this study, we mainly focused on the Android legacy[1] and new[2] applications written in Scala and Kotlin, respectively. At the time of writing, both of the applications are actively employed by end users as Wire is in a transitional period. In the long term, it is expected that Wire will phase out the legacy version of the application.

A critical entity of the ecosystem in study is the company that runs and develops the Wire messenger which, by overload, we also call "Wire". Wire manages two crucial components; the *distribution server*, denoted by S_{DS}, and the *ACME server*, denoted by S_{ACME}. Communication between a client

[1] https://github.com/wireapp/wire-android
[2] https://github.com/wireapp/wire-android-reloaded

and any of the servers is secured using a client-to-server protocol that we specify in Section 3.2. The distribution server is mainly responsible for relaying E2E-encrypted messages from a user's client to another user's client. Since MLS, the distribution server also acts as the orchestrator server, resolving conflicting MLS `Commit` messages, as discussed in Section 3.7.3. Other responsibilities of this server will be unveiled as we proceed into this Chapter. Note that we frequently use the term server as a shorthand reference to the distribution server. The ACME server is a new addition to the ecosystem and it is responsible for issuing x509 certificates to clients required the MLS end-to-end protocol. This server is only useful for MLS and irrelevant to Proteus. A more in depth discussion about the ACME server can be found in Section 3.7.3. When we refer to Wire as an entity, we refer to either the company, the application, the distribution or the ACME servers. Which entity we refer to will be clear from context.

3.2 Client-to-Server Protocol

A Wire client establishes a secure channel with the servers in order to prevent potential network adversaries from tampering with the communication between them, while simultaneously protecting the exposed metadata in E2EE messages. Recognizing the importance of this concern, Wire aligns with established best practices for secure messaging platforms by choosing Transport Layer Security (TLS) as its designated C2S protocol. TLS is an extensively studied and formally analyzed protocol that is proven to deliver strong security guarantees.

3.2.1 Server Authentication

During the TLS handshake, the server always authenticates to the client using a certificate that binds the server's identity to its public key. The client will then use the system-installed trusted certificates to verify that at least one of them vouches for its legitimacy by verifying the received certificate chain. Typically, there are hundreds of certificates installed on a system that are trusted by default and owned by hundreds of Certificate Authorities (CAs). A CA is an entity responsible for issuing certificates and performing the appropriate checks during the issuance of certificates, e.g., verifying possession of the private key corresponding to the public key and control of the domain in a certificate request. Hence, the ultimate trust is placed in CAs. It is sufficient to lose all the guarantees provided by TLS when at least one of them misbehaves. In practice, there were many incidents in the past [2].

Wire employs certificate pinning to protect against a misbehaving CA who issues a malicious certificate for Wire's servers with the ultimate goal of per-

forming a man-in-the-middle (MitM) attack. Certificate pinning is an additional security check to the basic certificate validation, which assures the server's authenticity to the client. Certificate pinning consists in the validation of the server's certificate by a client application, checking the certificate provided by the server during the handshake against the certificate stored ("pinned") within the client application.

3.2.2 Higher Level User Authentication

In TLS, client authentication is optional. Wire does not authenticate clients during the TLS handshake, which is common in many TLS applications. In contrast, client applications are required to prove that they are acting on behalf of a user on every protected RESTful API request. An example of such a request is during the client registration described in Section 3.3.2. We highlight that clients, as described in Section 3.1, never authenticate to the server; instead, authenticating as a user U is sufficient to act as any of its clients U_i at the C2S protocol level.

We now describe the details of user authentication on every request to the server. The API authentication mentioned above is based on a combination of the following:

1. *Short-lived access tokens* are used to authenticate to the protected API resources. These tokens are included in every HTTP request's authorization header field. This type of authentication is called Bearer authentication, also known as token authentication. An access token for user U is denoted by token_U.

2. *Long-lived user cookies* are used to continuously obtain new access tokens (sent from the server as HTTP cookies). A user cookie for user U is denoted by cookie_U.

Wire's long-lived user cookies are obtained on successful user login or after the completion of user registration described in Section 3.3.1. On the other hand, access tokens are refreshed in frequent time intervals using the aforementioned user cookies. The high level intuition of this authentication mechanism is that only the entity knowledgable of the user's credentials can obtain such tokens.

The content for both the user cookies and access tokens is almost identical. Below we present the most important fields they share in common.

- signature: An Ed25519 signature protecting the rest of the token's content using a server's signature key. The signature prevents the forgery of access tokens. Upon receiving a token, the server first verifies the signature in it.

- timestamp: A POSIX timestamp indicating the expiration time of the token.
- uuid: The 16-byte long user identifier of the user authenticated by this token.

3.3 Registration Protocol

Registration to Wire's messaging system enables end users to participate in conversations, exchanging messages with other registered end users. The registration protocol comprises two distinct processes: one for the registration of users and another for clients. It is important to reiterate the distinction between users and clients, as they serve distinct roles. A client is associated with a sole user, whereas a user can possess as many as seven clients. The integration of these two registration procedures, discussed in the subsequent sections, forms the foundation for multi-device support, which we believe to have strongly influenced Wire's design decisions. Our presentation begins with a description of user registration, followed by client registration.

3.3.1 User Registration

When an user U wishes to use Wire for the first time, they need to first register to Wire's messaging system. A diagram for the user registration procedure is provided in Figure 3.2.

It is worth noting that Wire distinguishes itself by not mandating user registration from a specific platform, setting it apart from other secure messaging services like WhatsApp, which necessitates registration through a mobile phone.

User registration begins with user U sending an email address email_U to server S_{DS}.To validate ownership of the provided email address, S_{DS} generates a random 6-digit verification code ($\mathsf{CodeGen}()$) and transmits it via email to the provided address. U fetches the verification code and sends it alongside an account password[3] (pw_U) and the email. Subsequently, S_{DS} verifies the match between the emailed and submitted verification codes (with a tolerance of up to three attempts; after three unsuccessful attempts, the user has to repeat registration). Upon successful validation, the server produces a user identifier (using $\mathsf{UUIDGen}()$) that functions as a distinct identifier in the system. The server generates a user cookie (using $\mathsf{CookieGen}(\cdot)$), setting the uuid field of the generated cookie to the newly generated user identifier ID_U. Last, the user identifier and cookie are returned to U.

[3]Wire's password policy mandates a password of at least 8 characters long of which at least 1 is lowercase, uppercase, digit and special characters.

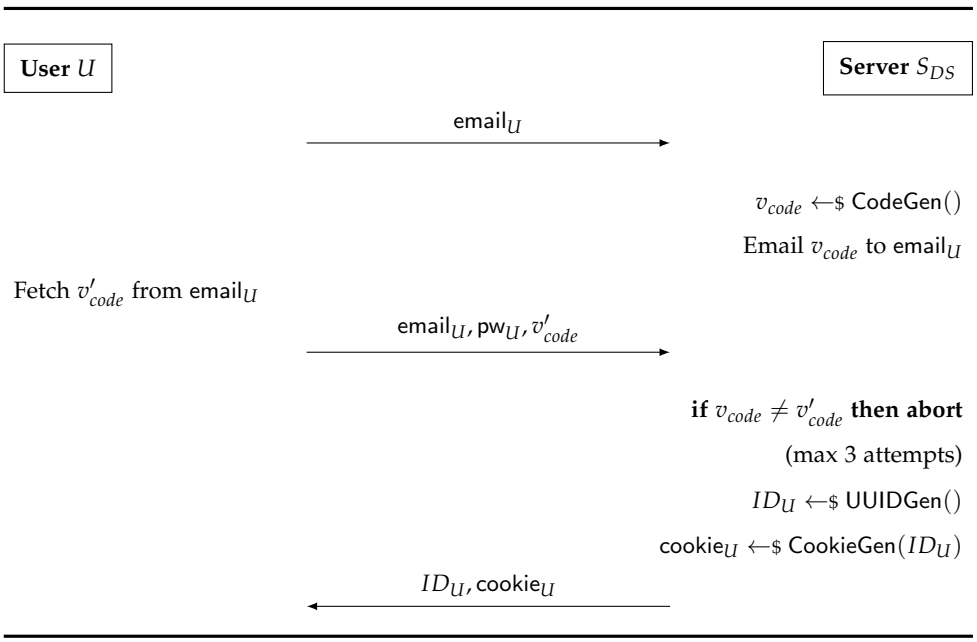


Figure 3.2: User Registration Protocol. We omit the specific details of how the server manages and persistently stores the information received and generated for the newly created user.

Note that user registration does not involve the generation of any long-term cryptographic key pairs. As a result, simply registering the user is not sufficient for E2E communication and at least a client must be registered alongside it. Wire's responsible entities for E2E communication are clients which can be registered at any later stage. To enable user interaction in conversations, user registration is always followed by client registration.

3.3.2 Client Registration and Multi-Device Support

Multi-device support poses a notable challenge for secure messaging platforms, a situation that persisted without comprehensive solutions for quite some time within major secure messengers such as WhatsApp, which previously required that the primary client had to be online for the secondary clients to send and receive messages. Wire presents a very simplistic approach as an attempt to solve this problem.

Wire's proposed solution involves a client registration procedure, shown in Figure 3.3. Unlike other secure messaging platforms, Wire does not feature a primary client to whom other clients can be linked. To put it succinctly, the access token token_U for a user U, also allows modifying the set of client identifiers associated with U. This set of clients is stored on the server S_{DS} and we denote it by $\mathcal{C}_U^{S_{DS}}$.

We will see that, when a client A_i belonging to user A wants to send a mes-

sage to another user B, it needs to access an updated collection of clients associated with another user B. This can be achieved through two distinct approaches. The first involves including modifications to $\mathcal{C}_U^{S_{DS}}$ in the server's messages when dispatching encrypted messages to a group, as elaborated in Section 3.6.4. In the second mechanism, A_i requests client identifiers for user B directly from the server S_{DS}, which returns an updated set in response. A_i replaces its existing set with the updated version and undertakes the requisite steps to integrate new clients within the underlying groups of conversations both client's users participate, as outlined in Section 3.6.4. Note that the exact details of client addition in MLS remain uncertain at the time of writing (either clients in a group will propose the addition, or the new clients will add themselves with external commits). The set of clients known to client A_i for user B is denoted by $\mathcal{C}_B^{A_i}$.

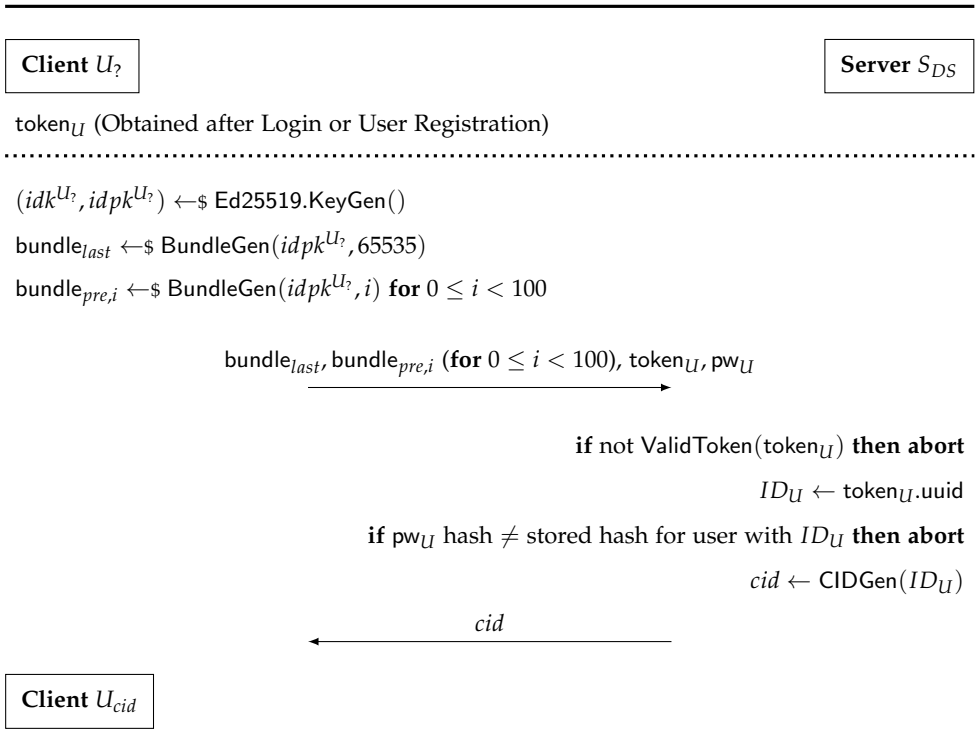


Figure 3.3: Client Registration Protocol. The diagram does not show how the server uses, manages and persistently stores the information received and generated for the newly created client. Moreover, client information, such as the type of device the client resides on, sent during registration is omitted.

Client registration is depicted in Figure 3.3. Notably, the client does not yet possess an identifier, indicated by the question mark in our notation. We assume that a client $U_?$ intending to register possesses an access token for user U. Subsequently, the client generates a long-term Ed25519 identity

key pair through the corresponding key generation algorithm. Afterwards, the client generates a set of 100 *pre-key bundles*, as well as a *last resort bundle* (both using $\mathsf{BundleGen}(.,.)$) (see Section 3.6.2 for more details). Conceptually, these bundles encapsulate the long-term public identity key of the client and pre-computed key pairs to be employed during session initialization; an in-depth inspection of pre-key bundles and session establishment is provided in Section 3.6. The client then forwards these bundles alongside the access token and the password associated with user U. The last two are used to validate that the registering entity corresponds to U. Initially, the access token undergoes verification utilizing the public key associated with S_{DS}'s signing key (using the $\mathsf{ValidToken}(.)$ function). The server extracts the user identifier from the token, uses the identifier to retrieve the corresponding password hash from the database, and confirms whether the hash of the provided password aligns with the stored hash. In the event of a match, the server proceeds to generate a new client identifier (using $\mathsf{CIDGen}(.)$)., which must be unique with respect to the user's existing clients. The client identifier is then returned to the client, officially establishing its status as a client of user U. Note that details related to token validation, hash computation and client identifier generation are omitted, as they are not needed. Moreover, we emphasize that clients of the same user are not required to interact during client registration. Each client acts, in essence, independently.

3.4 Multi-End-to-End Protocol Support

Wire has long provided support for Proteus, enabling communication between users in conversations. However, in recent years, Wire has taken a step forward by initiating the integration of MLS E2E group messaging protocol. This addition enables multi-protocol support and the coexistence of groups operating with either MLS or Proteus. In this section, we describe how Wire has organized various cryptographic components and libraries to realize their vision.

3.4.1 Cryptographic Components Hierarchy

The cryptographic components and libraries used/developed by Wire are organized into three levels, as shown in Figure 3.4. We list them starting from the lower level and moving to the higher level.

1. The *protocol level* consists of the low-level protocol implementations.
    a) The Proteus[4] component is maintained and developed by Wire using the Rust programming language. It is an implementation variant of Signal's protocol and Sesame session management.

[4] https://github.com/wireapp/proteus

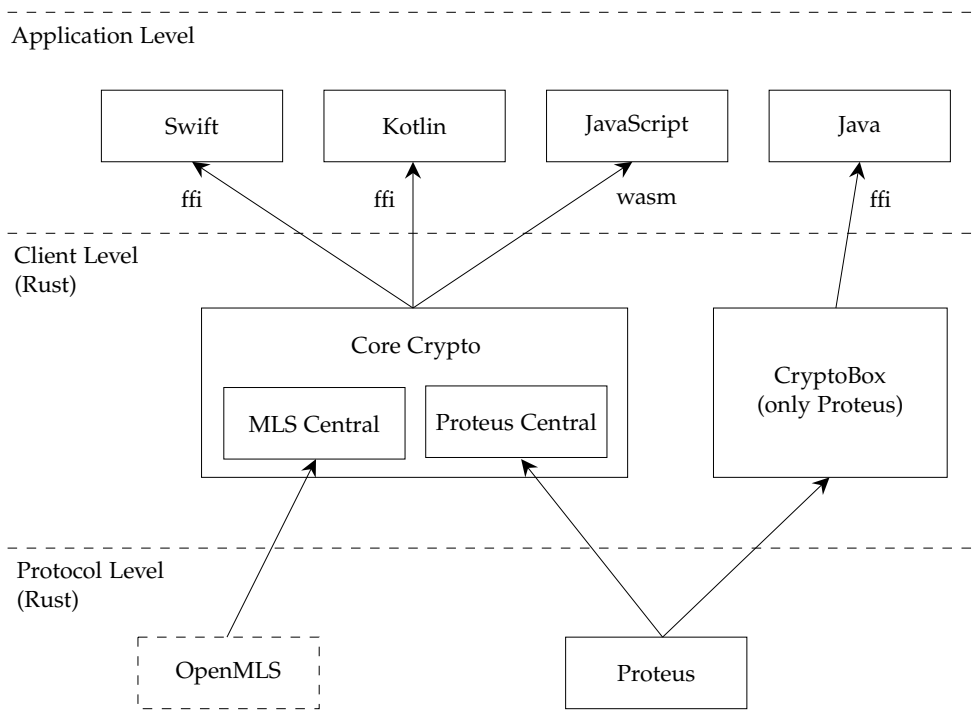


Figure 3.4: Cryptographic Components Hierarchy. Components in dashed box are not developed by Wire.

Thus, it provides the functionalities for Proteus sessions and dialogues explained in Sections 3.6.2 and 3.6.3, respectively, without persistent storage.

b) The OpenMLS[5] component is an open-source implementation in Rust of the MLS protocol. It is maintained and supported by Phoenix R&D and Cryspen, and it provides an interface for accessing MLS groups' operations while simultaneously hiding their complexity. For instance, it handles any modifications to the group's state; the result of additions, removals, and updates for clients, and performs any required validation steps.

2. The *client level* has components that utilize the interfaces provided by the protocol-level components in order to provide higher-level APIs with persistent storage.

   a) The CryptoBox[6] component supports only the Proteus protocol. It provides a higher-level Rust API that enables accessing Proteus dialogues (the session management protocol) but not sessions directly. It also provides persistent storage for all dialogues a client

[5] https://github.com/openmls/openmls
[6] https://github.com/wireapp/cryptobox

has and all necessary cryptographic keys, namely long-term and pre-key pairs.

b) The Core Crypto[7] component abstracts differences between MLS and Proteus, implementing a common interface for the two components. This allows to transparently use either Proteus or MLS behind the same API. The MLS Central provides an MLS-specific high-level API with persistent storage for MLS groups and other related information, such as encryption and signature key pairs and credentials. The Proteus Central is very similar to CryptoBox. We highlight that MLS groups are managed at the Core Crypto level, while Proteus groups are managed at the application level.

3. At the *application level*, we find the different programming languages for which Wire generates bindings of the high-level APIs from the client level. The legacy Android application uses the CryptoBox bindings to Java. For iOS and the new Android applications, there are Swift and Kotlin bindings of Core Crypto. Finally, Wire produces a wasm binary of Core Crypto for support in JavaScript for the web and desktop (based on Electron) applications.

3.5 Message Types and Serialization

Wire applications can comprehend and process only certain types of messages. These messages are what will eventually be encrypted using one of the E2E protocols and decrypted by the recipients. The recipients of these messages will then process and display them in a conversation visible to the user. These messages are serialized and deserialized using Protocol Buffers [1]. In general, every message has the following fields:

- A message identifier that can be used to reference the message in other messages.
- A message type.

We now list the most relevant to this study message types:

- `Text`: A text message with some string content. The recipient displays the content in a conversation.
- `Confirmation`: This can be either a delivery or read confirmation. The former confirms the delivery of a message to a client (the receiver must send a different `Confirmation` message to each of the user's clients) even if the user did not read the message displayed in a conversation; this can happen when the application is running in the background.

[7] https://github.com/wireapp/core-crypto

The latter occurs exclusively when the user reads the message in a conversation; that is, it is visible on the screen. The confirmed messages are referenced using their identifiers. The recipient visually marks the referenced messages as delivered and/or read.

- `ResetClientAction` (only in Proteus): Informs the recipient client that a new dialogue was initialized on the sender's side, which will result in the initialization of a new opaque Proteus session. The recipient displays a message in a conversation informing the user about the underlying Proteus session addition, but the client does not take any further action.
- `Ephemeral`: A message with an expiration time. It can be a `Text` message or any of the other supported types not specified here. The recipient will display the message which will disappear after the specified time interval.

Comprehensive details concerning all message types are documented within Wire's .proto specification file[8].

3.6 End-to-End Protocol – Proteus

Developed internally by Wire, Proteus draws significant inspiration from Signal's X3DH key exchange and the Double Ratchet protocol for establishing sessions among clients. Furthermore, Proteus adopts a session management protocol closely resembling that of the Sesame algorithm [44], also designed by Signal. In this section, we delve into the details of Proteus sessions and dialogues and how their interaction is used to build group messaging. Additionally, we look closely at the client applications, the communication with the distribution server and the information the server manages. We also highlight the exposed functionalities accessible at the application level, as depicted in Figure 3.4, which is very important for our analysis in Chapter 4 because E2E channels between clients are realized as Proteus dialogues, which manage multiple sessions. Only dialogues' functionality is exposed to the application; hence, we study security properties such as FS and PCS at the dialogue level of abstraction.

3.6.1 Notation - Proteus Session

The notation used in the context of Proteus sessions is summarized below:

- $ps^{tag}_{A_i \leftrightarrow B_j}$: A Proteus session between clients A_i and B_j, where A_i is the initiator. $tag \in \{0,1\}^{128}$ is a label (unique within a Proteus dialogue)

[8] https://github.com/wireapp/generic-message-proto/blob/master/proto/messages.proto

assigned to this Proteus session. The session maintained at client A_i (resp. B_j) is denoted by $ps^{tag}_{A_i \to B_j}$ (resp. $ps^{tag}_{A_i \leftarrow B_j}$).

- $(idk^{U_i}, idpk^{U_i})$: U_i's Curve25519 long-term identity key pair.
- $(ebk^{U_i}, ebpk^{U_i})$: U_i's Curve25519 ephemeral base key pair.
- $(prek^{U_i}_x, prepk^{U_i}_x)$: U_i's x^{th} Curve25519 pre-key pair, where $x \in \{0,1\}^{16}$. When $x \neq 65535$ it is ephemeral otherwise short-term.
- $(lrk^{U_i}, lrpk^{U_i})$: U_i's Curve25519 short-term last resort pre-key pair. It is equivalent to the $(prek^{U_i}_{65535}, prepk^{U_i}_{65535})$ pre-key pair.
- $(rchtk^{U_i}_x, rchtpk^{U_i}_x)$: Curve25519 ratchet key pair, used to asymmetrically ratchet forward to U_i's x^{th} sending symmetric ratchet in a Proteus session (which Proteus session is involved will be clear from the context).
- ms: master secret derived from XPDH in a Proteus session (which Proteus session is involved will be clear from the context).
- $rk^{U_i}_x$: root key used to asymmetrically ratchet forward to U_i's x^{th} sending symmetric ratchet in a Proteus session (which Proteus session is involved will be clear from the context).
- $ck^{U_i}_{x,y}$: chain key used to derive the y^{th} message key and the $(y+1)^{th}$ chain key on U_i's x^{th} sending symmetric ratchet in a Proteus session (which Proteus session is involved will be clear from the context).
- $(enck^{U_i}_{x,y}, mack^{U_i}_{x,y})$: message key (encryption and MAC keys) used to encrypt the y^{th} message sent on U_i's x^{th} sending symmetric ratchet in a Proteus session (which Proteus session is involved will be clear from the context).

3.6.2 Proteus Sessions

The establishment of a Proteus session between two clients is necessary for E2E communication in Wire's Proteus groups. Notably situated at the protocol level of abstraction within Wire's cryptographic libraries, as illustrated in Figure 3.4, Proteus sessions remain inaccessible directly to client's high level application interfaces. Wire applications only use Proteus sessions implicitly, through the dialogue abstraction, as elaborated in Section 3.6.3.

Broadly, the lifecycle of Proteus sessions consists of four distinctive operations: registration, initialization, asymmetric ratcheting, and symmetric ratcheting. During registration a client publishes to the server keying material essential for other clients to initialize a session with them. This is done in order to support key exchange when one of the communicating clients is offline. Initialization is responsible for the establishment of a session between

two participating clients by deriving a shared master secret. Subsequently, the processes of asymmetric and symmetric ratcheting are employed to derive new secrets within sessions, ensuring the sessions uphold advanced security properties, namely Forward Secrecy and Post-Compromise Security.

Proteus sessions offer the capability for asynchronous message exchange, facilitated by the presence of the distribution server that permits offline communication by caching messages destined for clients that are not connected to the network at the moment the messages are sent. Consequently, the Proteus session states at both communicating clients might not be synchronized. Furthermore, in order to accommodate out-of-order message delivery, a Proteus session necessitates the retention of intermediate message keys and other secrets for yet-to-be-received messages.

With this foundation in place, we begin to clarify the various elements that make up a Proteus session. For the rest of this section, we assume two clients denoted by I and R, representing the initiator and responder clients (of not necessarily distinct users), respectively.

Pre-key Bundles and Registration

As detailed in Section 3.3.2, Wire's client registration process is the operation of registering a client within the messaging system. This operation transmits the keying material needed for session establishment to the distribution server, by aggregating all essential information for a single session initialization into what is called a pre-key bundle. Precisely, a pre-key bundle generated by a client R (using $\mathsf{BundleGen}(idpk^R, pkid)$), includes the following fields:

- $pkid \in \{0,1\}^{16}$: The *pre-key identifier*. This attribute uniquely identifies the pre-key bundle and serves to correctly retrieve the corresponding pre-key pair from storage during the initiation of a session as the responder.
- $idpk^R$: The public long-term identity key for R. Note that the server will not complain in case a client uploads pre-key bundles with different public identity keys.
- $prepk^R_{pkid}$: The public pre-key of the newly generated pre-key pair $(prek^R_{pkid}, prepk^R_{pkid}) \leftarrow\!\!\$\ \mathsf{Ed25519.KeyGen}()$. Notably, the private key is absent from the pre-key bundle.
- (Optional) $\sigma \leftarrow \mathsf{Ed25519.Sign}(idk^R, prepk^R_{pkid})$: Representing a signature of $prepk^R_{pkid}$ present in the bundle signed with idk^R, the private key corresponding to the public long-term key in this bundle.

The pre-key pair associated with each pre-key bundle is stored in the local storage of R and is identifiable using the pre-key identifier.

After registration, the Wire client periodically uploads its bundles to the distribution server, replenishing the set of bundles when their number is below a certain threshold and are close to being exhausted. This frequent uploading stems from the fact that pre-key bundles are designed for a single use, with the special case of the last resort bundle. This particular last resort bundle serves the function of key-exchange material when the distribution server exhausted its ephemeral pre-key bundles for a particular user. The pre-key identifier 65535 is reserved for the last resort bundle. Unlike the standard ephemeral pre-key bundles, the last resort bundle can be used on multiple session initializations, requiring periodic updates within short intervals to achieve certain security goals, namely to limit the number of sessions affected when corruption occurs. The versions of Wire for Android do not update the last resort bundle; we discuss the security implications of this omission in Section 4.4.5.

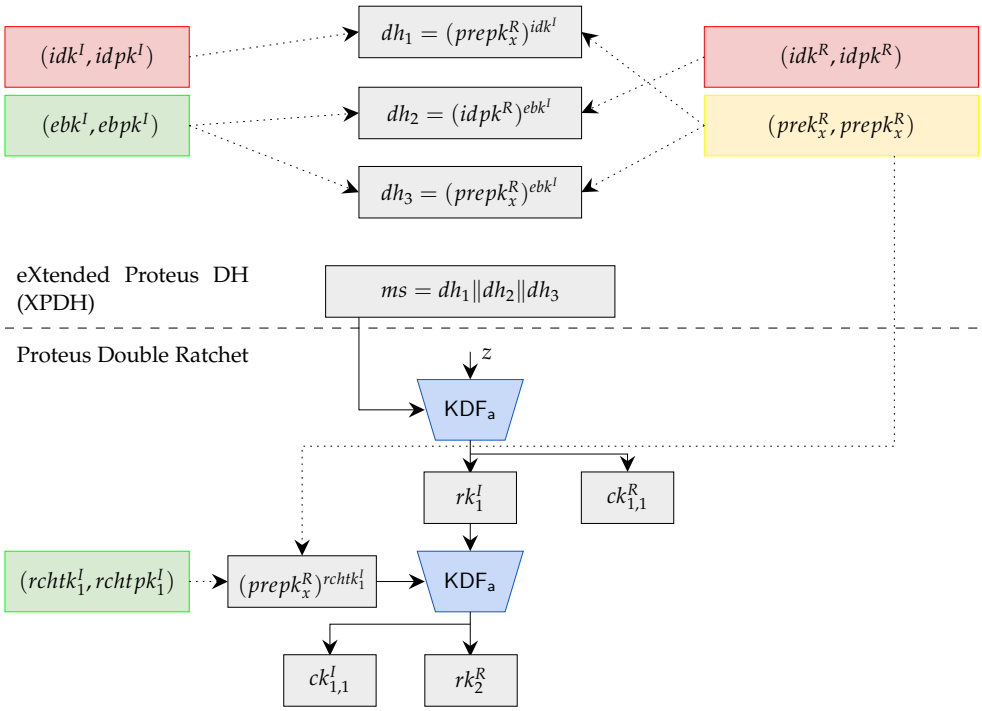
Moreover, as listed above, pre-key bundles have an optional signature the presence of which is not enforced neither by the distribution server nor the client applications. To be more precise, Wire's server and applications do not attempt to verify this signature, even if it is provided. In contrast, Signal requires signature for the short-term key which is always used in the initial key exchange as shown in Figure 3.5b. The implications arising from this differentiation are explained in Section 4.4.2.

Initialization and eXtended Proteus Diffie-Hellman (XPDH)

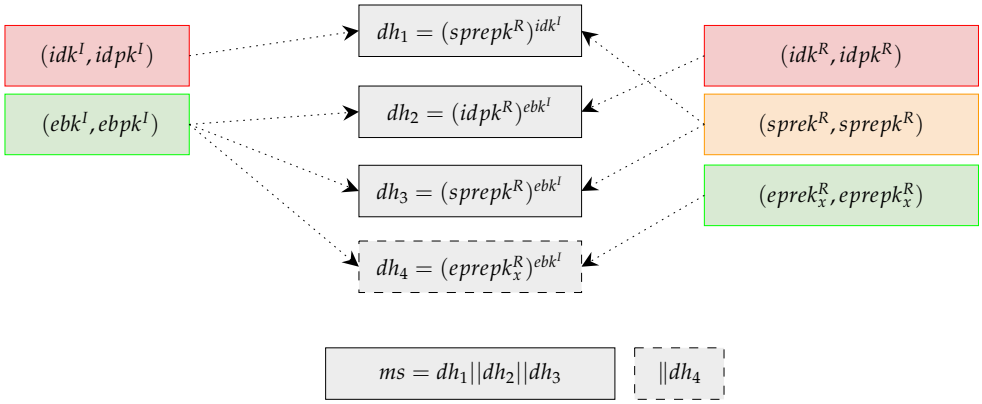
We now describe how a Proteus session $ps^{tag}_{I \leftrightarrow R}$ between the initiator client (I) and the responder client (R) is initialized. The initiator and the responder clients may belong to different users, or to the same user.

The clients execute a key exchange protocol to establish the initial shared secret between the communicating clients. This key exchange protocol is based on Signal's X3DH protocol, with certain differences in how asymmetric keys are combined in Diffie-Hellman computations to derive the initial shared secret. We named the key exchange protocol involved in Proteus's session initialization the eXtended Proteus Diffie-Hellman (XPDH). X3DH and XPDH are graphically depicted in Figure 3.5.

Initialization from the Initiator's Perspective The initiator client (I), associated with a long-term identity key pair $(idk^I, idpk^I)$, retrieves a pre-key bundle for the responder client (R) from the server. This bundle can be an ephemeral pre-key bundle or the last resort bundle. Let's assume the retrieved bundle holds the pre-key identifier $x \in \{0,1\}^{16}$. Notably, the online



(a) eXtended Proteus Diffie-Hellman (XPDH) initial key exchange and first (special) asymmetric ratchet step.



(b) eXtended Triple Diffie-Hellman (X3DH) initial key exchange

Figure 3.5: Initial key exchange between the Initiator client (I) and Responder client (R). The DH computations are done from the point of view of the Initiator (I). Key pairs with boxes colored are ephemeral, are sometimes ephemeral (pre-key other than last resort key) and others short-term (last resort key), are always short-term, are long-term. The last DH shared value dh_4 in X3DH is optional indicated by dotted border. z is the all zero input.

presence of the responder client is not needed; the server maintains pre-computed bundles for clients to enable offline communication. This eliminates the need for one client to wait for another's online status in order to initialize a session. Subsequently, the initiator calculates the value:

$$ms \leftarrow (prepk_x^R)^{idk^I} \| (idpk^R)^{ebk^I} \| (prepk_x^R)^{ebk^I}$$

where ebk^I is the private key of a freshly generated key pair $(ebk^I, ebpk^I) \leftarrow\$$ Ed25519.KeyGen(). On a high level, the $(prepk_x^R)^{idk^I}$ value serves to authenticate the initiator to the responder, $(idpk^R)^{ebk^I}$ authenticates the responder to the initiator, and $(prepk_x^R)^{ebk^I}$ guarantees Session Independence. XPDH achieves FS immediately if the bundle used was ephemeral. In case the last resort bundle was used then a client achieves FS only when it updates its last resort bundle.

Following the derivation of the shared master secret, the initiator client asymmetrically ratchets forward twice in order to derive the chain key for its first sending symmetric ratchet. The resulting Proteus session for the initiator is denoted by $ps_{I \rightarrow R}^{tag}$, for a randomly chosen tag. The details of the XPDH protocol from the perspective of the initiator are shown in Figure 3.5a.

Lastly, I will eventually use the established session to encrypt the first batch of messages to R. Proteus sessions employ an AEAD scheme for encrypting and decrypting messages. The associated data (AD) of the first encrypted messages (the `PreKeyMessage`) of I will contain the pre-key identifier x, the public ephemeral base key $ebpk^I$, and the public long-term key $idpk^I$ that I used during XPDH, as shown in Figure 3.8. This allows the responder to also recover the master secret and, thus, complete session initialization on both sides.

Initialization from the Responder's Perspective The responder client (R) associated with the long-term identity key pair $(idk^R, idpk^R)$ gathers the necessary information to derive the same shared secret. More specifically it retrieves the pre-key identifier x, the public ephemeral base key $ebpk^I$, and the public long-term key $idpk^I$ from the associated data of the encrypted messages (`PreKeyMessage` messages) sent by I. Using the pre-key identifier, the responder recovers the pre-key pair from local storage and proceeds to compute the shared secret as follows:

$$ms \leftarrow (idpk^I)^{prek_x^R} \| (ebpk^I)^{idk^R} \| (ebpk^I)^{prek_x^R}$$

The resulting Proteus session for the responder is denoted by $ps_{I \leftarrow R}^{tag}$.

Asymmetric and Symmetric Ratcheting/Steps

Two fundamental constituents within a Proteus session are the asymmetric and symmetric ratcheting operations, which together constitute the Double

Ratchet protocol illustrated in Figure 3.6, which is similar to Signal's Double Ratchet protocol with some minor differences on the KDFs used. On a high level, the *symmetric ratcheting* operation in Proteus is performed when a client receives encrypted messages sent on an already existing receiving symmetric ratchet or when a client sends new messages in the session. On the other hand, *asymmetric ratcheting* comes into play whenever a client receives an encrypted message from the other party within a new receiving symmetric ratchet.

We describe, without loss of generality the asymmetric ratcheting mechanism for I. I will either:

1. Generate a fresh key pair $(rchtk_x^I, rchtpk_x^I) \leftarrow\$ \mathsf{Ed25519.KeyGen}()$. This scenario arises when I needs to initialize a new sending symmetric ratchet for sending a new message. The DH share $(rchtpk_x^R)^{rchtk_x^I}$ combines the newly generated key pair $(rchtk_x^I, rchtpk_x^I)$ with the public ratchet key $rchtpk_x^R$ of the responder utilized in the previous asymmetric step.

2. Use the public ratchet key $rchtpk_x^R$ obtained through an encrypted message of the new receiving symmetric ratchet. This scenario occurs when I needs to initialize a new receiving symmetric ratchet in response to a message from R. Here the DH share $(rchtpk_x^R)^{rchtk_{x-1}^I}$ combines the received public ratchet key $rchtpk_x^R$ with the ratchet key pair used in the preceding asymmetric step $(rchtk_{x-1}^I, rchtpk_{x-1}^I)$, which the initiator previously generated.

In both cases, these DH shares are subsequently used as inputs for $\mathsf{KDF_a}$, alongside the root key rk_x^I (in the first case) or rk_x^R (in the second case) derived from the preceding asymmetric step. The output of this asymmetric step is a fresh root key (rk_{x+1}^R in the first case and rk_x^I in the second case) and chain key ($ck_{x,1}^I$ in the first case and $ck_{x,1}^R$ in the second case). The new chain key marks the beginning of a new symmetric ratchet. In-depth details into $\mathsf{KDF_a}$ are provided in Figure 3.7a.

The aforementioned chain key serves as the cornerstone for starting a new symmetric ratchet. The application of $\mathsf{KDF_m}$ with chain key $ck_{x,y}^I$, as shown in Figure 3.7b, derives encryption key $enck_{x,y}^I$ and MAC key $mack_{x,y}^I$ required for the AEAD scheme Proteus uses. We collectively refer to these keys as the message key. At the same time, this application of $\mathsf{KDF_m}$ advances the symmetric ratchet forward by a single step as a new chain key $ck_{x,y+1}^I$ is simultaneously created.

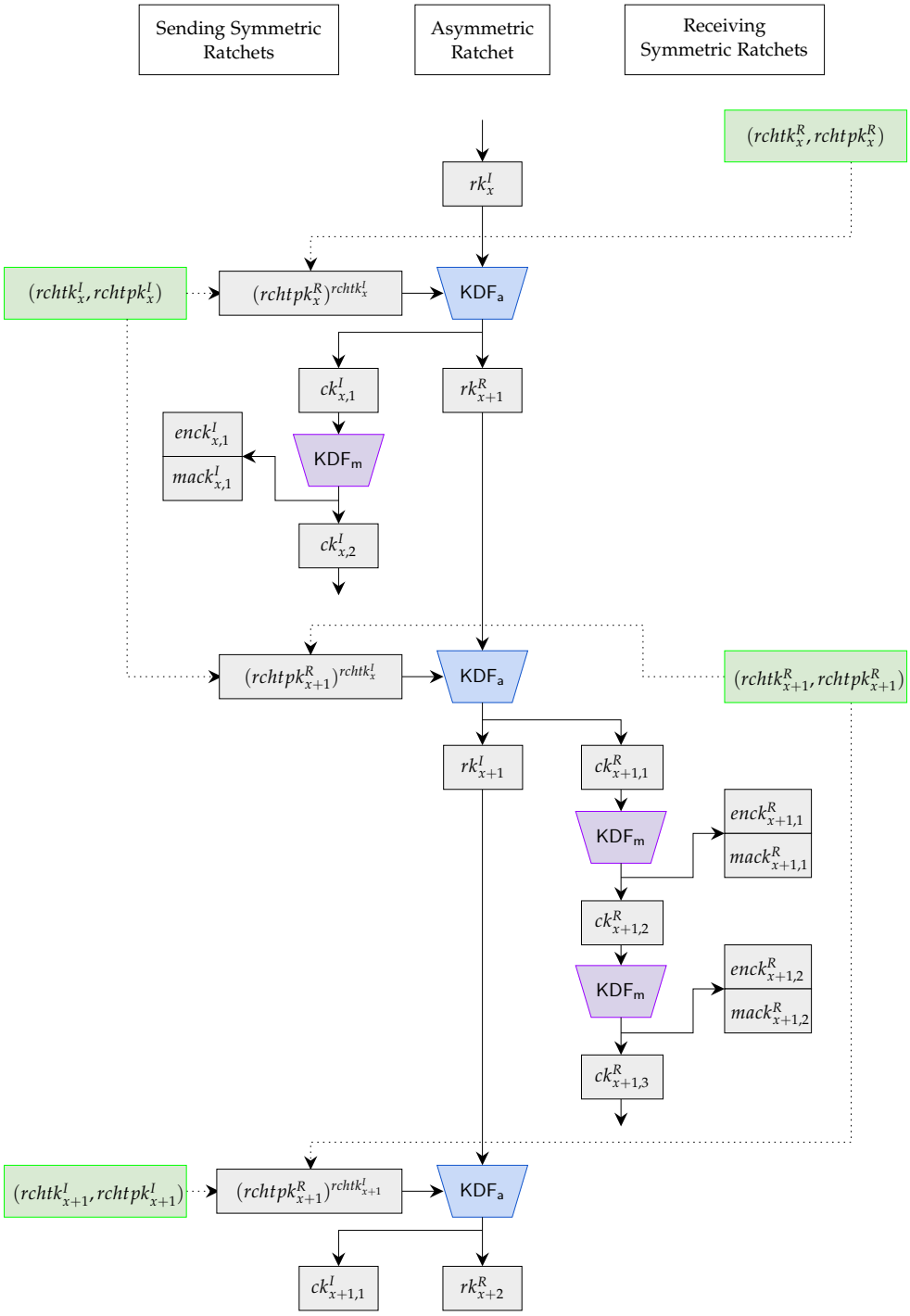
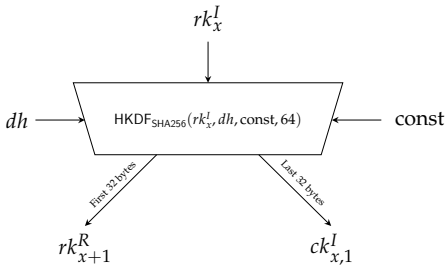
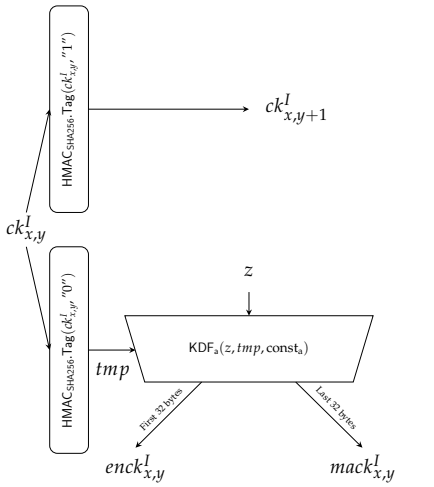


Figure 3.6: Proteus Double Ratchet Protocol. The diagram is based on the point of view of the Initiator client (I): DH computations and references to sending and receiving symmetric ratchets. Key pairs with boxes colored are ephemeral. Symmetric steps are colored, while asymmetric are colored.



(a) $\mathrm{KDF_a}$, the KDF used for asymmetric ratchet updates, dh is the shared DH value computed from the ratchet keys designated for this stage. The constant value const is one of "handshake" or "dh_ratchet". The former is used only for the first asymmetric ratchet step after XPDH.



(b) $\mathrm{KDF_m}$, the KDF used for symmetric ratchet updates, z is the all zero input and const_a is equal to "hash_ratchet".

Figure 3.7: Proteus Double Ratchet Key Derivation Functions (KDFs).

Proteus Session Encryption

Let's consider a Proteus session $ps^{tag}_{I\to R}$. We denote the encryption algorithm applied within this session by $ps^{tag}_{I\to R}.enc(m)$, where m is the plaintext being encrypted. The plaintext message m in the context of Wire conforms to the types listed in Section 3.5, which Wire applications expect. We emphasize that the Proteus session encryption is not directly accessible at the application level shown in Figure 3.4, the protocol-level library will first select which Proteus session to use in a dialogue. We discuss Proteus dialogues in Section 3.6.3.

We now describe, without loss of generality, the Proteus session encryption algorithm of the y^{th} message within the most recent (assume x^{th}) sending symmetric ratchet of the initiator I. The algorithm starts with a step in the symmetric ratchet of the client's most recent sending ratchet: the only sending ratchet maintained in a Proteus session (only the chain key $ck^I_{x,y}$ result of the last symmetric advancement of the ratchet and the ratchet key pair $(rchtk^I_x, rchtpk^I_x)$ used for the asymmetric step that resulted into $ck^I_{x,1}$ are stored) in order to achieve FS.

The symmetric advancement yields a fresh chain key $ck^I_{x,y+1}$ maintained for the next symmetric step, in addition to a message key employed for AEAD encryption. The message key comprises of a 32-byte encryption key $enck^I_{x,y}$, utilized for ChaCha20 encryption of message m (with the nonce equating the message number on the sending ratchet), and a 32-byte MAC key $mack^I_{x,y}$, to authenticate the associated data (sent in plaintext) as well

as the resultant ciphertext from ChaCha20 encryption. After the symmetric step on the sending ratchet of the client, the previous chain key $ck^I_{x,y}$ and the message key ($enck^I_{x,y}$, $mack^I_{x,y}$) used for encryption are erased from storage in order to provide FS. The AEAD encryption of the y^{th} message within the x^{th} sending symmetric ratchet of client I, using the derived message key, is visually represented in Figure 3.8. Below, we list the fields present in an encrypted message:

- τ: The HMAC-SHA256 tag ensuring the integrity of associated data and ciphertext.
- type $\in \{1, 2\}$: A one byte value indicating the presence of the optional fields in the associated data. When type $= 1$ the optional fields are present and the encrypted message is called a `PreKeyMessage`; otherwise, a `CipherMessage`.
- (Optional) pkid: Pre-key identifier of the pre-key bundle used by the initiator during XPDH.
- (Optional) $ebpk^I$: The ephemeral public base key generated by the initiator during XPDH.
- (Optional) $idpk^I$: The long-term public identity key of the initiator.
- tag: The Proteus session tag utilized for identifying a session managed by a Proteus dialogue.
- y: The nonce given to the ChaCha20 encryption algorithm, equivalent to the number of messages encrypted using the x^{th} sending symmetric ratchet of I thus far.
- ctr_{prev}: The number of messages encrypted during the previous sending symmetric ratchet of *InitiatorClient*, which can be used to signal its "end".
- $rchtpk^I_x$: The public ratchet key used for asymmetric ratcheting forward to the symmetric ratchet responsible for deriving the encryption keys for this particular message.
- ctxt: The encrypted message content.

The optional fields are only included during the first sending symmetric ratchet of the initiator client (I). A `PreKeyMessage` is the sole method to disseminate the pre-key identifier pkid, the public ephemeral base key $ebpk^I$, and the public long-term key $idpk^I$ necessary for performing XPDH on the responder's end, thereby enabling the derivation of the same initial shared secret. We also highlight that Proteus session encryption does not perform asymmetric steps.

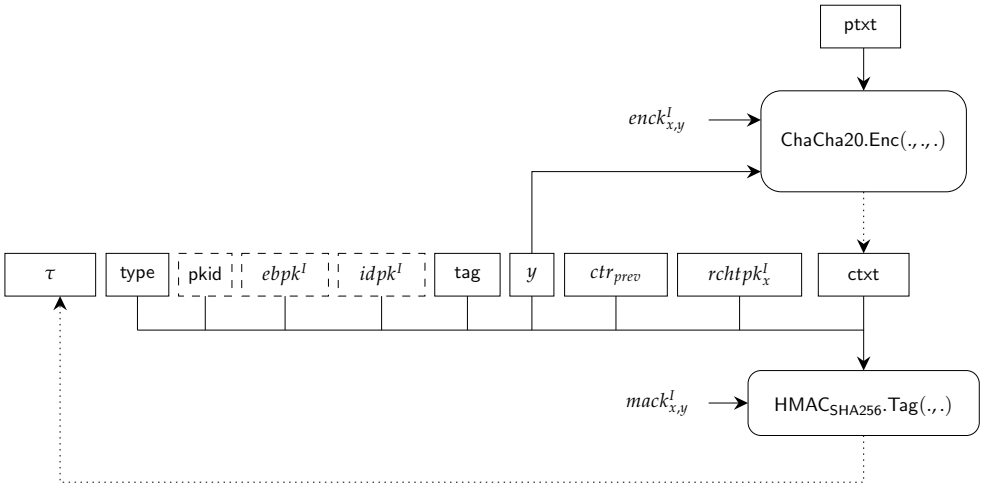


Figure 3.8: Encryption of y^{th} message on the x^{th} sending ratchet of the initiator and resulting message content. The dashed fields are optional and required only when the initiator client encrypts messages on its first sending ratchet ($x = 1$). The presence of the optional fields is indicated by the type field.

Proteus Session Decryption

The asynchronous nature of the messaging system can lead to the reception of encrypted messages from the counterpart client out of order. Wire's proposed solution to this challenge, which is implemented in Proteus sessions, does not mandate synchronization (reception of messages in the expected order) but allows multiple symmetric steps of the receiving symmetric ratchets when the communicating clients desynchronize, i.e., when a message in the far future is received. Additionally, the session can decrypt messages on previous receiving ratchets.

The above-mentioned solution necessitates maintaining essential secrets, as well as message keys, in the local storage for both previous and current receiving ratchets. Indeed, Proteus sessions maintain the five most recent receiving symmetric ratchets, each storing up to 1000 message keys, along with the chain key from the ratchet's last symmetric advancement and the public ratchet key of the other client used for the asymmetric step that resulted to the symmetric ratchet. In general, the decryption algorithm is responsible for both asymmetric ratchet steps and symmetric steps on the receiving symmetric ratchets. This differs from the encryption algorithm, which only performs symmetric steps within the latest sending symmetric ratchet. Note that decryption always prepares the ground for encryption as it asymmetrically ratchets forward twice, aligning with the subsequent sending symmetric ratchet as we explain below.

Let us consider a Proteus session $ps^{tag}_{I \to R}$. We denote the decryption algo-

rithm applied within this session by $ps^{tag}_{I\to R}.dec(c)$, where c is the encrypted message. We emphasize that the Proteus session decryption, similarly to encryption, is not directly accessible at the application level shown in Figure 3.4, but only through a Proteus dialogue.

The entirety of the decryption algorithm operates as a transaction, ensuring that changes to the Double Ratchet are applied only in the absence of errors. If an error occurs while processing a message, the ratchet state is reset to the state before the message was processed. Additionally, when the generation of new message keys exceeds the available buffer space for a particular receiving ratchet, the algorithm will automatically evict the oldest message key in a first-in-first-out (FIFO) manner, making room for the new keys. When an entire receiving ratchet is evicted, all message keys, the chain key, and the public ratchet key stored for it are removed.

Let's assume without loss of generality that $ps^{tag}_{I\to R}$ is currently at the x^{th} sending symmetric ratchet with ratchet key pair $(rchtk^I_x, rchtpk^I_x)$. It follows that the last five receiving symmetric ratchets stored in the session are the x^{th}, $(x-1)^{th}$, ..., $(x-4)^{th}$, accompanied by corresponding public ratchet keys $rchtpk^R_x$, ..., $rchtpk^R_{x-4}$. The decryption algorithm has two steps.

First Step Upon receiving an encrypted message, the first step involves determining whether an asymmetric ratchet step is necessary or if the receiving ratchet required for decryption is already available in the session's storage. This can be done by comparing the value of the received public ratchet key $rchtpk^R_{x'}$, which is part of the associated data of the encrypted message c, with the value of $rchtpk^R_r$ for $x-4 \leq r \leq x$ (the session's locally stored public keys); here, the public ratchet key serves as an identifier used for lookup in the stored receiving ratchets. If a match is found, the corresponding receiving symmetric ratchet is chosen for decryption. If not, the algorithm executes two asymmetric ratchet steps. The initial asymmetric step invokes the $\mathsf{KDF_a}$ function as follows:

$$rk^I_{x+1}, ck^R_{x+1,1} \leftarrow \mathsf{KDF_a}(rk^R_{x+1}, (rchtpk^R_{x'})^{rchtk^I_x}, \text{"dh\_ratchet"})$$

This computation results in the first chain key $ck^R_{x+1,1}$ of the receiving symmetric ratchet used for encrypting the received message. Subsequently, the second asymmetric step invokes $\mathsf{KDF_a}$ once again:

$$rk^R_{x+2}, ck^I_{x+1,1} \leftarrow \mathsf{KDF_a}(rk^I_{x+1}, (rchtpk^R_{x'})^{rchtk^I_{x+1}}, \text{"dh\_ratchet"})$$

In this context, $rchtk^I_{x+1}$ denotes the private key of a newly generated key pair $(rchtk^I_{x+1}, rchtpk^I_{x+1}) \leftarrow\$ \mathsf{Ed25519.KeyGen}()$. This key pair and the chain key $ck^I_{x+1,1}$ characterize the new sending symmetric ratchet and are sufficient for future symmetric steps. The previous sending symmetric ratchet of I is effectively replaced by the new $(x+1)^{th}$ ratchet (assuming decryption terminates without errors).

Second Step Upon completion of the first step, let us assume that its output is the i^{th}, where $x - 4 \leq i \leq x + 1$, receiving symmetric ratchet with latest chain key $ck^R_{i,j}$ and a set of stored message keys denoted by $\mathsf{mk\_store}_i$. Given this information, the decryption algorithm can proceed with the decryption of the encrypted message using the AEAD scheme used by Wire. However, prior to decryption, the appropriate message key must be obtained for the task. This step is divided into two scenarios: either the message key has already been derived from a previous decryption and resides within $\mathsf{mk\_store}_i$, or a number of symmetric steps within the i^{th} receiving symmetric ratchet is required. In order to decide, we use the y field from the associated data of c, which indicates the index of the message key in the receiving symmetric ratchet used for encrypting the message.

When $y < j$ (first scenario), the algorithm searches for the message key with index y within $\mathsf{mk\_store}_i$. If the message key $(enck^R_{i,y}, mack^R_{i,y})$ is found, it is subsequently used to verify the authenticity of the encrypted message ($\mathsf{HMAC}_{\mathsf{SHA256}}.\mathsf{Vfy}(mack^R_{i,y}, \tau, \mathsf{ad}\|\mathsf{ctxt})$, where $\tau\|\mathsf{ad}\|\mathsf{ctxt} = c$ and ad is the associated data of c), followed by its decryption ($\mathsf{ChaCha20}.\mathsf{Dec}(enck^R_{i,y}, y, \mathsf{ctxt})$). In case the verification process fails, an `InvalidSignature` error is returned. Conversely, if the message key is not found, two potential error scenarios arise: an `Outdated` error is the result if the oldest message key $(enck^R_{i,k}, mack^R_{i,k})$ within $\mathsf{mk\_store}_i$ is more recent than the targeted message key ($j < k$); otherwise, a `Duplicate` error occurs.

When $y \geq j$ (second scenario), the algorithm advances the i^{th} receiving symmetric ratchet the required number of times; specifically, the difference $y - j$ plus one step for the derivation of the message key, unless $y - j > 1000$ where a `TooFarDistant` error is raised. With each advancement, the derived message keys, excluding the target key used for decryption, are stored in $\mathsf{mk\_store}_i$. In contrast, only the last chain key $ck^R_{i,y+1}$ is kept in storage to strengthen Forward Secrecy. The target message key derived from this process $(enck^R_{i,y}, mack^R_{i,y})$ is first used for decrypting the received message ($\mathsf{ChaCha20}.\mathsf{Dec}(enck^R_{i,y}, y, \mathsf{ctxt})$), subsequently validating the authenticity of the encrypted message ($\mathsf{HMAC}_{\mathsf{SHA256}}.\mathsf{Vfy}(mack^R_{i,y}, \tau, \mathsf{ad}\|\mathsf{ctxt})$). Notice that, decryption precedes verification in this scenario, whereas an AEAD encryption scheme following the Encrypt-then-MAC construction employes verification first. Luckily for Wire, verification immediately follows decryption and since ChaCha20 decryption seems not to offer any side channel, we believe that this construction cannot be exploited (e.g. mounting a padding oracle attack).

We would also like to emphasize that the ctr_{prev} field in the associated data is not taken into account during decryption, thus, the end of the previous receiving symmetric ratchet is not specified. Therefore, it can be extended

even after ratcheting forward to the next receiving symmetric ratchet. The PCS consequences of this omission are discussed in Section 4.4.5.

3.6.3 Proteus Dialogues

At a higher level, the communication among clients takes place through the framework of Proteus dialogues. A dialogue is an abstraction that we introduce to model application level behavior. Proteus dialogues and Signal's Sesame protocol both internally handle sessions at the protocol Level. The main differences are that Sesame maintains sessions of the Signal protocol and new sessions are periodically created/inserted every hour, while Proteus dialogues manage Proteus sessions and only manual dialogue resets create/insert sessions.

At its core, a Proteus dialogue, denoted by $\mathcal{D}_{B_j}^{A_i}$, takes the responsibility of overseeing the 99 most recent Proteus sessions established at A_i (the local client) in the context of communication with B_j (the remote client). If more than 99 sessions are to be managed then the oldest will be removed, following a FIFO policy. We say that a Proteus session belongs to a dialogue if and only if it is managed by the dialogue and we denote it by $ps_{A_i \to B_j}^{tag} \in \mathcal{D}_{B_j}^{A_i}$.

Conceptually, when A_i wish to dispatch an E2E message to B_j, it relies on the encryption functionality provided by $\mathcal{D}_{B_j}^{A_i}$. This internal process effectively selects one of the Proteus sessions it manages between the two parties to encrypt the intended message. We call this Proteus session the active session for the dialogue which we denote by $\mathcal{D}_{B_j}^{A_i}.\mathsf{active}$. Similarly, when A_i wants to decrypt an E2E message received from B_j, it again relies on the decryption provided by dialogue $\mathcal{D}_{B_j}^{A_i}$. The dialogue, during decryption, either selects an existing Proteus session or initiates a new one to perform the decryption of a received encrypted message.

The details of Proteus dialogue initialization, encryption, and decryption are given in the following discussions. Notably, it is important to remember that only the dialogue functionality is exposed to the application level (as illustrated in Figure 3.4). Consequently, this design makes Proteus sessions completely opaque from the direct control of clients.

Dialogue Initialization

A dialogue is responsible for managing and utilizing sessions to enable message encryption and decryption. As a basic requirement for E2E communication, at least one Proteus session is needed within a Proteus dialogue. Consequently, when initializing a dialogue, a new Proteus session is created.

More precisely, Proteus dialogues handle sessions between two clients, each identified by their long-term identity key pairs. It is necessary that Proteus dialogues maintain consistency by ensuring that all Proteus sessions that they manage are initialized (as described in Section 3.6.2) using the same identity keys for both the local client and the remote peer. This *consistency rule* prevents an adversary from inserting a Proteus session with an adversary-controlled identity key pair in the Proteus dialogue. In order to respect the consistency rule, a Proteus dialogue maintains the identity keys used during the XPDH initial key exchange of its first Proteus session (for future validation and session setups).

We now describe the two options available for initializing a dialogue $\mathcal{D}_{B_j}^{A_i}$.

Initialization from a Pre-Key Bundle Given a pre-key bundle for client B_j, a Proteus session $ps_{A_i \to B_j}^{tag}$ is initialized as described in Section 3.6.2, with initiator client A_i. The tag (tag) given to the session is randomly chosen from $\{0,1\}^{128}$. A dialogue $\mathcal{D}_{B_j}^{A_i}$ is returned with $ps_{A_i \to B_j}^{tag} \in \mathcal{D}_{B_j}^{A_i}$ and $\mathcal{D}_{B_j}^{A_i}.\mathsf{active} = ps_{A_i \to B_j}^{tag}$.

Initialization from a PreKeyMessage Given an encrypted message c originating from a client B_j, with whom we don't have a Proteus dialogue yet, the following procedure is followed. If the message c is a `CipherMessage`, an `InvalidMessage` error is returned and Proteus dialogue initialization fails. This result occurs due to the requirement of `PreKeyMessage` to provide the necessary information (in the associated data) to initialize a session. Conversely, if c is a `PreKeyMessage`, fields from the associated data used for initialization are extracted, namely the pre-key identifier (pkid), the long-term public identity key ($idpk^{B_j}$), and the ephemeral public base key ($ebpk^{B_j}$) used by the remote client. The pkid is used to recover the pre-key pair from the local storage. The absence of the requested pre-key pair (from the storage) leads to a `PreKeyNotFound` error. However, in cases where the pre-key pair is present, a Proteus session is initialized as described in Section 3.6.2, with responder client A_i. Note that the tag (tag) associated with the new session is determined by the remote client and is found in the associated data of c.

Having successfully initialized the session $ps_{B_j \leftarrow A_i}^{tag}$, the next step involves decrypting the received `PreKeyMessage` c , $m \leftarrow ps_{B_j \leftarrow A_i}^{tag}.dec(c)$. If decryption results into an error, the same error is returned, leading to the failure of dialogue initialization. Conversely, successful decryption to a non-error message m indicates that both parties derived the same secrets, an implicit form of authentication since the initiator demonstrates correct derivation of the DH shared secret, and initialization is deemed successful. If $\mathsf{pkid} \neq 65535$ (different than the last resort key pair identifier), the pre-key

employed for session initialization is removed from the local storage. Finally, the result is the first decrypted message m and a Proteus dialogue $\mathcal{D}_{B_j}^{A_i}$ with $ps_{B_j \leftarrow A_i}^{tag} \in \mathcal{D}_{B_j}^{A_i}$ and $\mathcal{D}_{B_j}^{A_i}.\mathsf{active} = ps_{B_j \leftarrow A_i}^{tag}$.

Dialogue Encryption

Given a Proteus dialogue $\mathcal{D}_{B_j}^{A_i}$, encrypting a message m to B_j is as simple as encrypting using the Proteus dialogue's active Proteus session. We denote Proteus dialogue encryption by $\mathcal{D}_{B_j}^{A_i}.enc(m) = \mathcal{D}_{B_j}^{A_i}.\mathsf{active}.enc(m)$. We highlight that client applications can only encrypt messages using this functionality.

Dialogue Decryption

The decryption functionality provided by Proteus dialogues, denoted by $\mathcal{D}_{B_j}^{A_i}.dec(c)$ for dialogue $\mathcal{D}_{B_j}^{A_i}$ is responsible for determining the appropriate Proteus session that should be used to decrypt a received encrypted message c. Furthermore, it is responsible for the task of initializing new Proteus sessions when the managed sessions are unable to decrypt the received encrypted message, provided the message permits session initialization; specifically, if it is a `PreKeyMessage`. Once again, we highlight that only the decryption functionality of Proteus dialogues is available at the application level.

Given a Proteus dialogue $\mathcal{D}_{B_j}^{A_i}$, decrypting an encrypted message c originating from a client B_j, apply the procedure below. (In what follows, without loss of generality, assume that A_i is the responder in all sessions)

- If c is a `CipherMessage` and $\nexists ps_{B_j \leftarrow A_i}^{tag'} \in \mathcal{D}_{B_j}^{A_i}$ such that $tag' = tag$, where tag is the session tag extracted from the associated data of c, then the dialogue issues an `InvalidMessage` error (a message was received but its session tag does not match any session in the dialogue). Conversely if such a session $ps_{B_j \leftarrow A_i}^{tag}$ exists, it is employed for decryption, resulting in $m \leftarrow ps_{B_j \leftarrow A_i}^{tag}.dec(c)$. If the decryption encounters an error, the error is returned; otherwise, the decrypted message m is returned.
- If c is a `PreKeyMessage`, the dialogue performs the consistency rule sanity check which compares the remote client long-term public identity key in the associated data of c with the one maintained in the dialogue $\mathcal{D}_{B_j}^{A_i}$. If these public keys fail to match, a `RemoteIdentityChanged` error is returned. If there is no error and
  - $\exists ps_{B_j \leftarrow A_i}^{tag'} \in \mathcal{D}_{B_j}^{A_i}$ such that $tag' = tag$, where tag is the session tag extracted from the associated data of c, it is used to decrypt c,

resulting in $m \leftarrow ps^{tag}_{B_j \leftarrow A_i}.dec(c)$. The result of the decryption is returned, except when the `InvalidSignature` error occurs which is treated differently as we see below.

- $\nexists ps^{tag'}_{B_j \leftarrow A_i} \in \mathcal{D}^{A_i}_{B_j}$ such that $tag' = tag$ or the special case above of `InvalidSignature` error when $\exists ps^{tag'}_{B_j \leftarrow A_i} \in \mathcal{D}^{A_i}_{B_j}$ are encountered, then the dialogue attempts to initialize a new Proteus session $ps^{tag}_{B_j \leftarrow A_i}$. The session initialization and decryption procedure is identical to what we described, in an earlier discussion, in the dialogue initialization from a `PreKeyMessage` (Section 3.6.3). Successful decryption using the freshly established session $ps^{tag}_{B_j \leftarrow A_i}$ leads to its inclusion within the managed sessions of dialogue $\mathcal{D}^{A_i}_{B_j}$ (note that the oldest managed session is removed if the dialogue exits 99 sessions that it handles). We also highlight, since this is relevant for the attack in Section 4.3.1, that in case the new Proteus session has the same tag as any session already managed by the dialogue, the previous session is replaced by the new one. Finally, the result of the decryption is returned.

For both `CipherMessage` and `PreKeyMessage`, if decryption is succesful, the Proteus session $ps^{tag}_{B_j \leftarrow A_i}$ used for decryption becomes the new active Proteus session of the dialogue ($\mathcal{D}^{A_i}_{B_j}.\text{active} = ps^{tag}_{B_j \leftarrow A_i}$). This minor detail introduces notable security implications, which we will discuss in Section 4.3.

Dialogue Reset

Wire clients can reset Proteus dialogues. Users have to manually perform resets on their clients, typically, when they observe decryption errors or other error in conversations. When a client U_i, uses this feature on the Proteus dialogue $\mathcal{D}^{U_i}_{V_j}$ with another client, V_j, the following happens. First, U_i will destroy the previous dialogue and initialize a fresh one from a pre-key bundle of V_j. Note that destroying a Proteus dialogue only affects the dialogue $\mathcal{D}^{U_i}_{V_j}$ at U_i, not V_j; V_j will keep the dialogue $\mathcal{D}^{V_j}_{U_i}$ with its state, and keep using it to manage the previously established Proteus sessions. Subsequently, U_i encrypts a `ResetClientAction` message using the newly initialized dialogue, resulting in a `PreKeyMessage`. This `PreKeyMessage`, when received at V_j, leads to the initialization of a new Proteus session managed by dialogue $\mathcal{D}^{V_j}_{U_i}$. Moreover, the result of decrypting this message informs V_j about the underlying changes, which otherwise are completely opaque, but the Wire clients we analyze don't react to this information. Only a message in one of the user V's conversations will indicate this change.

We highlight that a Proteus dialogue $\mathcal{D}_{U_i}^{V_j}$ at client V_j can contain both sessions in which V_j is the initiator and sessions where U_i is the initiator. For instance, if V_j initializes the Proteus dialogue $\mathcal{D}_{U_i}^{V_j}$ from a pre-key bundle of U_i and later U_i resets its Proteus dialogue, as described above, then $\mathcal{D}_{U_i}^{V_j}$ will contain two sessions where in the first V_j is the initiator and in the second the responder (assuming now other actions where taken by the two clients).

3.6.4 Group Messaging

Until now, we have discussed the low-level details of Proteus sessions and the higher-level details of Proteus dialogues, an abstraction for E2E channels between two clients. However, in a messaging system like Wire, the need for varied group interactions between clients arises, where clients communicate simultaneously with many other clients (of possibly many users).

A Proteus group is an abstraction for a many-to-many clients E2E communication by combining pairwise Proteus dialogues between each client in the group. Also, clients can be part of multiple groups in many different combinations. Every communication in Wire is group communication using group messaging, even if only two clients are involved. Moreover, a group is always below a conversation, which is an abstraction for the communication between users by ignoring their clients involved in the group. In essence, a conversation is a chat accessible from a device's application interface where messages are displayed as messages sent from users, although clients sent them.

In the Proteus world, a conversation between users is denoted by $\mathcal{W}_{\mathsf{proteus}}^{gid}$, where $gid \in \{0,1\}^{128}$ is a unique 16-byte identifier for the conversation (inherited from the underlying group). We refer to the users in conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$ as the *members of the conversation*. The *conversation membership* $\mathfrak{W}_{\mathsf{proteus}}^{gid}$ is the set containing all members of $\mathcal{W}_{\mathsf{proteus}}^{gid}$. Similarly, the conversation's underlying group between clients is denoted by $\mathcal{G}_{\mathsf{proteus}}^{gid}$. We refer to the clients in group $\mathcal{G}_{\mathsf{proteus}}^{gid}$ as the *members of the group*. The *group membership* $\mathfrak{G}_{\mathsf{proteus}}^{gid}$ is the set containing all members of $\mathcal{G}_{\mathsf{proteus}}^{gid}$. Remarkably, the size of groups and conversations are not constrained; they could even consist of a single member. In order to learn the conversation membership information, clients send a request to the server, whose response includes the requested information. The client has to be a client of a user who is a member of the conversation in order to get access to the conversation membership. Recall that the access tokens authorize a client to act on behalf of a user.

Thus far, we have studied conversation membership in relation to users and how a client U_i can obtain this information from the server. However, users alone do not suffice for E2E communication since U_i must learn the mem-

bership of the conversation's underlying group for group messaging. To obtain the group membership $\mathfrak{G}^{gid}_{\mathsf{proteus}}$, client U_i simply combines the clients belonging to each member of $\mathcal{W}^{gid}_{\mathsf{proteus}}$ that U_i is aware of, given in the following expression:

$$\mathfrak{G}^{gid}_{\mathsf{proteus}} = \cup_{V \in \mathfrak{W}^{gid}_{\mathsf{proteus}}} \mathcal{C}^{U_i}_{V} \tag{3.1}$$

Once the client membership set $\mathfrak{G}^{gid}_{\mathsf{proteus}}$ is computed with Equation 3.1, the subsequent step involves initializing a Proteus dialogue, if one does not already exist, between the local client U_i and each client within this membership (pairwise dialogues are created between all the clients in a group). More specifically, if client U_i and V_j are members of the same group $\mathcal{G}^{gid}_{\mathsf{proteus}}$, U_i will first check if a dialogue with V_j exists in the persistent storage implemented and managed at the client level (of Figure 3.4) with an interface exposed at the application level, and only if such a Proteus dialogue does not exists U_i initializes a new dialogue from a pre-key bundle as described in Section 3.6.3.

The pairwise dialogues between all group members enable group messaging. Notably, the same Proteus dialogue serves as the means of E2E communication across all Proteus groups in which the participating clients share membership.

In the following paragraphs, we discuss group messaging which is implemented at the application level. We will examine the process of dispatching encrypted messages to the server, including additional metadata required by the server to forward messages to the intended clients. Furthermore, we explain how applications use the group/conversation identifier to ensure that they correctly assign messages for decryption to the relative groups and that messages are displayed in the correct conversation.

Encryption and Dispatch to Server

Let us consider a scenario where a user U, using client U_i, sends a message m within a Proteus group $\mathcal{G}^{gid}_{\mathsf{proteus}}$. The group encryption process steps are the following:

1. For each client $V_j \in \mathfrak{G}^{gid}_{\mathsf{proteus}} \setminus \{U_i\}$:
    a) If a Proteus dialogue $\mathcal{D}^{U_i}_{V_j}$ already exists at U_i, then retrieve it. If not, initialize the Proteus dialogue as elaborated in Section 3.6.3.
    b) Use the obtained Proteus dialogue $\mathcal{D}^{U_i}_{V_j}$ to encrypt the message m intended for V_j, resulting in $c_{V_j} \leftarrow \mathcal{D}^{U_i}_{V_j}.enc(m)$.

c) Prepare additional metadata $md_{V_j} = ID_V \| j$, consisting of the user identifier and the client identifier to uniquely specify the intended recipient client of c_{V_j}.

2. After encrypting the message m for every client $V_j \in \mathfrak{G}^{gid}_{\mathsf{proteus}} \setminus \{U_i\}$, send a request enclosing the mentioned additional metadata and the encrypted messages. Note that the server will also keep a UTC timestamp of the time it received the request. This request must also include:

   a) The identifier of the specific group to which the message is sent (gid).

   b) Sender information (ID_U, part of the access token, and i the client identifier) and access token for authorization checks.

3. After forwarding the request from step 2 to the distribution server S_{DS}, two potential responses may follow:

   - The server successfully received the request, containing all encrypted messages for every member of the group. In this case, the group encryption terminates.
   - Some encrypted messages are absent due to updates in a user's client list or conversation membership. The server informs U_i in the response about users (members of the conversation $\mathcal{W}^{gid}_{\mathsf{proteus}}$) whose certain clients lack encrypted messages. The server's response makes explicit the changes for both the conversation membership and the users' clients sets. Therefore, client U_i updates the sets of users' clients $\mathcal{C}_V^{U_i}$ for all users V in the server response. Then, U_i can update the conversation and group memberships by adding and removing members. Repeat from step 1, but in step 1 encrypt only to the newly added members to the group $\mathcal{G}^{gid}_{\mathsf{proteus}}$ and use in the request of step 2 the already encrypted messages and metadata from previous repetitions for the other members of $\mathcal{G}^{gid}_{\mathsf{proteus}}$.

The group encryption procedure ensures that a message is encrypted to all intended recipients, while also covering any dynamic changes to conversation membership and client lists of users.

Decryption and Message Rendering

After a client encrypts messages and dispatches them to the S_{DS}, the server takes the responsibility of relaying these encrypted messages to clients, possibly caching them until they are online and connect to the server.

Let us consider a scenario where a user V, using client V_j is about to receive an encrypted message c originating from U_i. Along with the encrypted data c, the server will also forward the following information:

- The unique identifier of the group (gid)
- The UTC timestamp indicating when the server received c (`timestamp`)
- Details about the sender (user and client identifiers)

Using the sender information, client V_j will either:

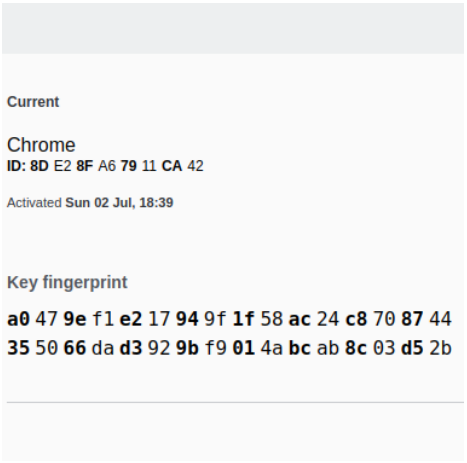
- if a Proteus dialogue $\mathcal{D}_{U_i}^{V_j}$ already exists, decrypt message c, yielding $m \leftarrow \mathcal{D}_{U_i}^{V_j}.dec(c)$.
- if no Proteus dialogue exists, the client attempts to initialize one by following the algorithm in Section 3.6.3. This initializes a new dialogue $\mathcal{D}_{U_i}^{V_j}$ and simultaneously, decrypts c to m.

Once decrypted, message m is displayed in the application interface of conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$ with identifier gid, with the display time determined by `timestamp`. Any decryption failures are displayed in this conversation and are visible to the end user (error message).

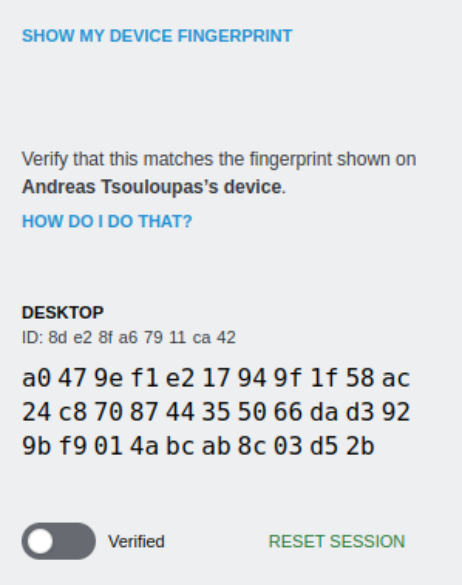
Group Verification Status

In order to prevent the distribution server from performing trivial impersonation attacks, clients must verify the authenticity of Proteus dialogues described in section 3.6.3. The verification procedure involves an out-of-band verification of the long-term public key of the remote client seen in the very first pre-key bundle or `PreKeyMessage` used to initialize the Proteus dialogue. More precisely, the two users of the clients involved have to meet each other in person and compare the fingerprints of the long-term public keys the clients have used to initialize the dialogue between them with the actual long-term public key fingerprints shown on the other user's client. The users must manually compare each of the 64 hex digits representing the long-term public identity key of the peer and then again manually switch the status of the dialogue to verified (Figure 3.9). The verification is essential since no certificates and a PKI are used to bind the public keys to an identity.

Generalizing the above to Proteus groups, a group $\mathcal{G}_{\mathsf{proteus}}^{gid}$ is deemed verified from the point of view of client U_i (member of the group) if and only if U_i verified all Proteus dialogues with the other client members of the group ($V_j \in \mathfrak{G}_{\mathsf{proteus}}^{gid}$), including other clients of user U.



(a) Screen of client being verified.



(b) Screen of verifier client.

Figure 3.9: Wire's desktop application screens for Proteus dialogue verification.

3.7 End-to-End Protocol - MLS

In addition to Proteus conversations, Wire is currently developing a new feature of MLS conversations. MLS is a newly standardized protocol that offers an efficient out-of-the-box protocol for asynchronous group messaging (based on tree structures) with FS and PCS in mind.

A group of clients aiming to exchange encrypted messages requires a method for deriving shared symmetric encryption keys. The challenge of enabling this secure E2E communication has been extensively explored for two-party scenarios, with the Double Ratchet emerging as a widely accepted solution. Channels that implement the Double Ratchet offer the benefits of fine-grained FS and PCS, but they achieve low communication efficiency in groups: the number of messages a client is required to encrypt (with different keys) is linear with the number of clients in the group. As described in Section 3.6.4, Wire uses the Double Ratchet-based Proteus sessions as a building block in Proteus dialogues between two clients to enable group messaging. Conversely, MLS groups achieve high communication efficiency: the number of messages a client is required to encrypt for group membership changes is logarithmic with the number of clients in the group, and for application messages it is one encryption.

In the following sections, we will provide an overview of MLS, followed by a more detailed exploration of key membership modification operations and its underlying tree structures. Additionally, we will delve into certain

decisions made by Wire that are left open in the MLS standard, decisions that are crucial for our analysis in Chapter 5. The details of the MLS protocol are documented in its standard [13].

Please note that the following sections do not aim to provide an exhaustive description of MLS. Instead, they offer a high-level overview to help us understand the fundamental concepts of MLS.

3.7.1 Overview

In MLS, clients are organized in groups, denoted by $\mathcal{G}_{\mathsf{mls}}^{gid}$, where $gid \in \{0,1\}^*$ is a unique arbitrarily long identifier for the group. The group's state in which a well-defined set of authenticated clients (the group members) hold shared cryptographic state is called *epoch*. Similarly to Proteus groups, we refer to the clients in group $\mathcal{G}_{\mathsf{mls}}^{gid}$ at a specific epoch $e \in \{0,1\}^{64}$ as the members of the group at e, and the membership $\mathfrak{G}_{\mathsf{mls}}^{gid,e}$ of group $\mathcal{G}_{\mathsf{mls}}^{gid}$ is the set containing all its members at epoch e. Each group is currently at a specific epoch, and all the epochs of the group are linearly connected, forming a sequence of its history. Clients within a group at a given epoch share an *epoch secret*. This secret is used to derive keying material used in encrypting and authenticating messages. Importantly, the membership of a group can change from epoch to epoch, and the protocol guarantees that clients who are not members of the group at epoch e cannot access the epoch secret at epoch e, even if they were members of the group in a previous epoch $e' < e$.

The three main components of MLS are:

- The *ratchet tree* specifies the group membership by maintaining at its leaves information for each group member. It serves to authenticate group members to each other and efficiently encrypt messages related to changes in the tree content for subsets of the group. Notably, each epoch has its distinct ratchet tree, which can vary in content (at tree nodes) and structure. Additionally, the protocol use the ratchet tree at a specific epoch to derive a *commit secret*, contributing to the derivation of the epoch secret.
- The *key schedule* defines the sequence of key derivations needed for advancing through epochs and deriving various other secrets. At each epoch, a fresh *encryption secret* is derived. This encryption secret is necessary for initializing the secret tree for that epoch.
- The *secret tree* uses the encryption secret from the key schedule to derive shared secrets for an epoch. These shared secrets enable group members to derive further keying material used in message encryption and authentication. Each member of the group is assigned two

symmetric ratchets, similar to what was described in Proteus symmetric ratchets. One ratchet is designated for `Handshake` messages (group state changes), while the other is for `Application` messages, such as text messages. All symmetric ratchets within the secret tree are accessible to all group members; hence, encrypting a `Handshake` or `Application` message to a group requires only one encryption. Note that each client signs every `Handshake` and `Application` message, allowing other client members to verify that the sender is authorized to use a particular symmetric ratchet for sending a message. The private signing key of each member is specified at the leaves of the ratchet tree, as seen in Figure 3.10. Ratchet trees are described in detail in Section 3.7.2

The connection between these three components is depicted in a high-level manner in Figure 3.10, which visually depicts the ratchet tree and the secret key of a group with four members at a certain epoch.

Changes to the ratchet tree are initiated by group members through `Proposal` messages that group members send to the group. There are three major types of proposals: (1) adding a client, (2) updating a client, and (3) removing a client. Detailed explanations of `Proposal` messages, the available types and their effect on the ratchet tree content (when applied in a `Commit`) are provided in Section 3.7.2. However, note that a `Proposal` message alone is not adequate to modify the ratchet tree. For a proposal to take effect, it must be included in a `Commit` message, which is then sent to the group by a group member. Once a group member receives and verifies a `Commit` message, all contained proposals are implemented and changes are reflected to the ratchet tree of the next epoch. Further details on `Commit` messages are provided in Section 3.7.2. These `Proposal` and `Commit` messages are jointly referred to as `Handshake` messages. Specific optional use cases exist that allow external clients (not members of the group) to propose changes or add themselves (through an external commit) to a group in the next epoch. These cases are not critical for understanding the main concepts of the MLS protocol and are therefore omitted.

Encryption and Authentication `Handshake` and `Application` messages can fall into two categories when sent as MLS messages:

- `PublicMessage`: This type of message exclusively includes a single `Handshake` message, which is not encrypted but is authenticated as being sent by a group member using a secret derived from the epoch secret of the current epoch.
- `PrivateMessage`: This type of message can contain both `Handshake` and `Application` messages (exactly one message). It is always encrypted and authenticated using message keys derived from the secret

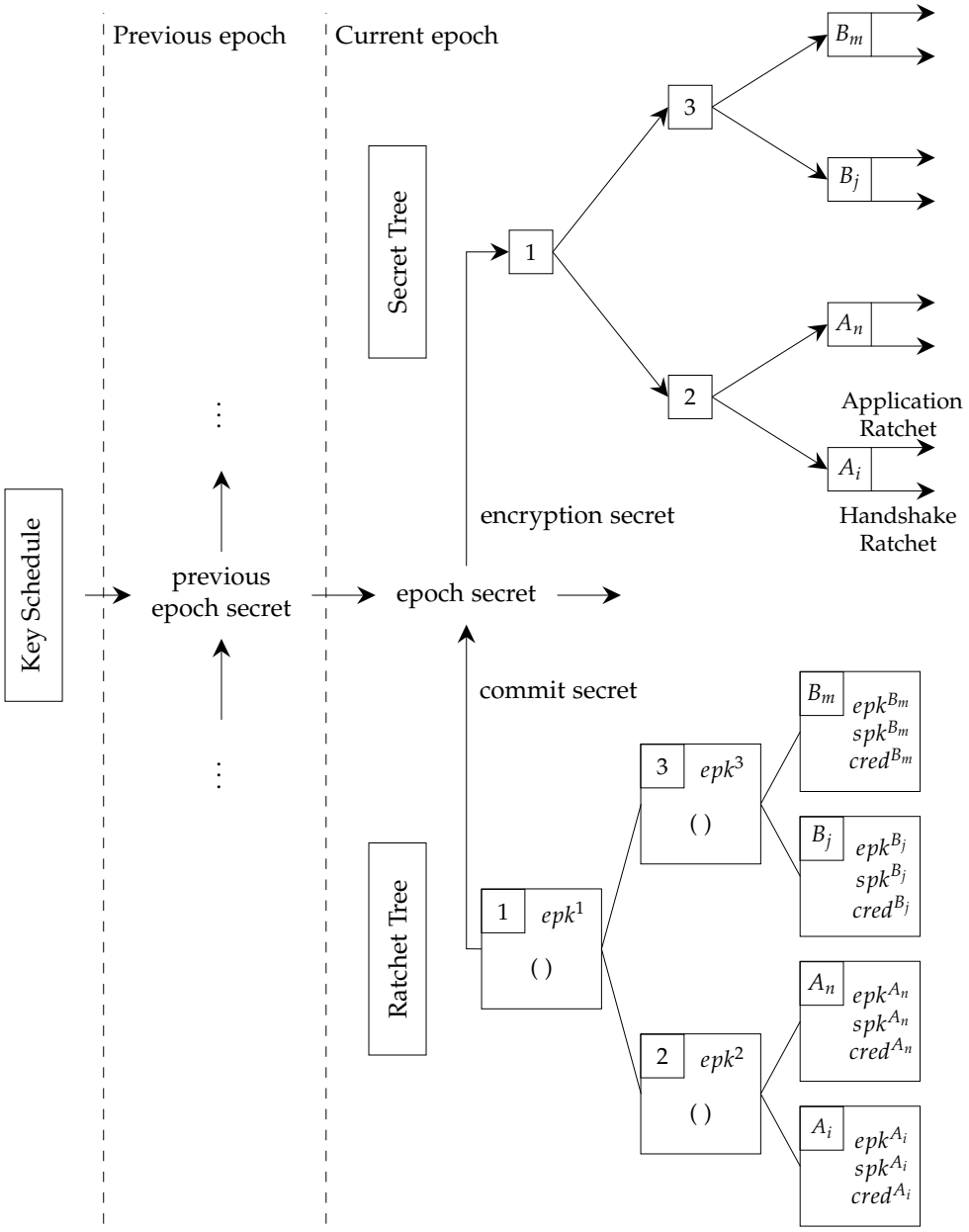


Figure 3.10: High-level view of the MLS components and their interactions in a group of four members: A_i, A_n, B_j and B_m.

tree either using a handshake symmetric chain or an application symmetric chain.

We note that both `PublicMessage` and `PrivateMessage` are additionally signed with the signing key of the sender client, as specified in the ratchet tree, making all messages non-repudiable. The specific cryptographic algo-

rithms used for key derivation, encryption, authentication, and signatures are given in the ciphersuite employed by the group. For simplicity, details of algorithms and how the ciphersuite is agreed are omitted here as we aim to only describe the core concepts of the MLS protocol rather than discuss the concrete instantiations.

MLS Services MLS assumes a trusted *Authentication Service* (AS) but a highly untrusted *Delivery Service* (DS). The AS allows group members to authenticate the credentials presented by other group members and, subsequently, their public signature key and identity. The DS is expected to reliably deliver messages to the clients participating in the MLS-enabled messaging system. In Wire, the distribution server S_{DS} takes the role of the delivery service while the ACME server S_{ACME} is the authentication service. Both these servers are depicted in Figure 3.1, showing the components of Wire's service.

3.7.2 The Ratchet Tree

The protocol uses the ratchet tree to describe the membership of a group in an epoch. Group modifications require only $\log(N)$ encryptions (exploiting the tree structure) using the stored key pairs at the nodes of the tree, instead of $N-1$ that would be needed to encrypt to every member (where N is the number of members in the group).

A ratchet tree is, in essence, a perfect binary tree, meaning it is a complete balanced binary tree. This type of tree has the property that if it has a depth d, then the number of leaves in the tree is exactly 2^d, which is the maximum number of leaves a binary tree of that depth can have. The leftmost leaf is assigned the index 0, while the rightmost leaf is assigned the index $2^d - 1$. The knowledge of a leaf's position is needed for removing a member from the group.

MLS assigns each group member to a leaf within the ratchet tree by including information of the member in the leaf node as we describe below. We note that not all leaves must be assigned a member; hence, they can be empty. Leaves, when assigned to a member, contain the following information:

- Public Encryption Key (epk^{U_i}): This key is used in hybrid public key encryption (HPKE) [14] to encrypt secrets associated with modifications in the tree's content, specifically to the member U_i assigned to the leaf. The corresponding private key (ek^{U_i}) is solely known to the client U_i.
- Public Signature Key (spk^{U_i}): This key is utilized for authenticating the member U_i to other group members. The corresponding private key

(sk^{U_i}) is known only to the client U_i.

- Credential ($cred^{U_i}$) – The credential can be employed to verify that the public signature key is bound to the identity of the member. MLS defines two types of credentials: x509 certificates, which bind an identity to the public signature key and can be verified using a root of trust, and basic credentials, which can represent an identity, with the public key being verifiable through out-of-band means.

Intermediate nodes, on the other hand, are all nodes in the tree that are not leaves. These nodes may have the following information:

- Public Encryption Key (epk^x) – Similar to leaves, this key is employed in HPKE to encrypt secrets related to changes in the tree's content. These secrets are intended for members who know the corresponding private key. A tree invariant ensures that the corresponding private key (ek^x) is exclusively known to members at leaves that are descendants of the intermediate node x. Here, x is the node's breadth-first (left to right) position, which is used solely for representation purposes. We emphasize that a leaf may not know a predecessor's private key.
- List of "Unmerged" Leaves – This list specifies the descendant leaves that do not know the private key ek^x associated with the intermediate node x.

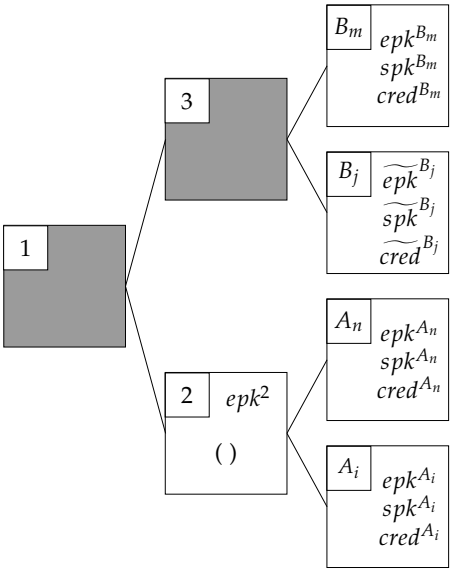
Both leaf nodes (when not assigned a member) and intermediate nodes have the potential to be empty, and these empty nodes are referred to as *blank* nodes.

Update Proposal

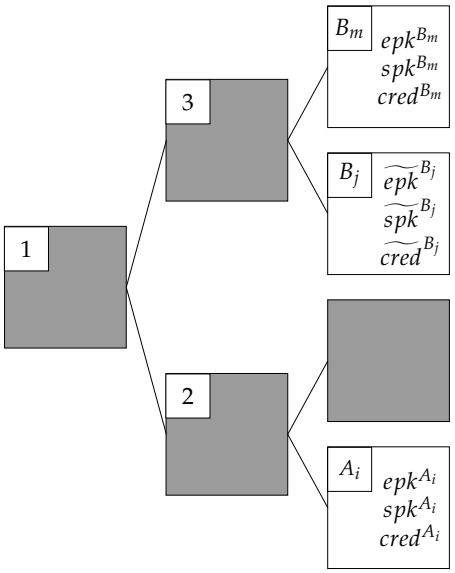
A client $B_j \in \mathfrak{G}_{\mathsf{mls}}^{gid,e}$ of group $\mathcal{G}_{\mathsf{mls}}^{gid}$ can request to modify the information stored at its assigned leaf within the ratchet tree using `Update` proposals. This `Proposal` message contains a fresh instance of the leaf node's contents, containing a new public encryption key ($\widetilde{epk}^{B_j}$), public signature key ($\widetilde{spk}^{B_j}$) and credentials ($\widetilde{cred}^{B_j}$). The processing of an `Update` (in a `Commit`) results in the following changes to the ratchet tree:

1. The content of the sender's (B_j's) leaf is substituted with the content provided in the `Update`.
2. All intermediate nodes encountered along the path from the sender's (B_j's) leaf (excluding the sender's leaf) to the root of the tree are blanked, meaning that the information they previously stored is emptied.

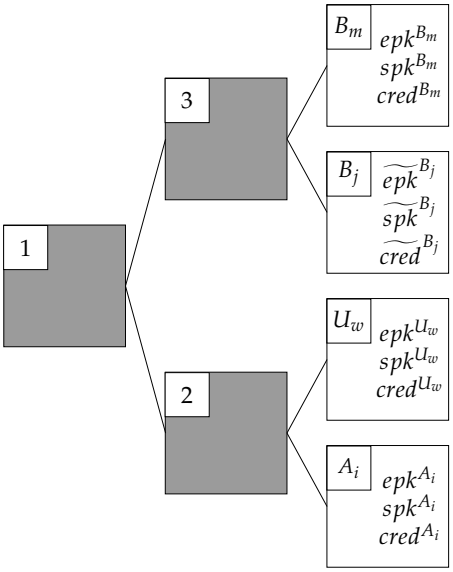
To illustrate this process, consider an initial ratchet tree as depicted in Figure 3.10. The modified ratchet tree after an `Update` is shown in Figure 3.11a.



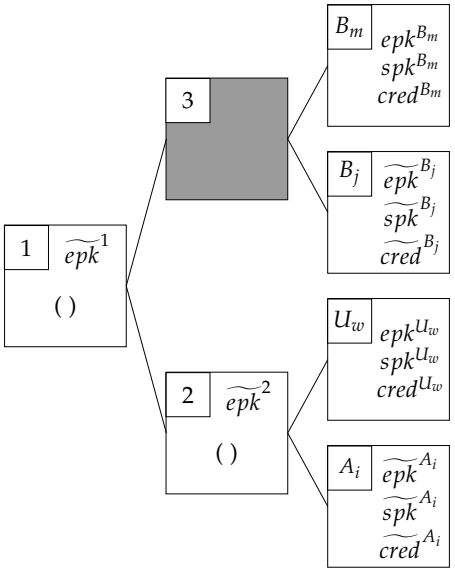
(a) Implementation of B_j's `Update` proposal.



(b) Implementation of A_i's `Remove` proposal removing A_n.



(c) Implementation of B_j's `Add` proposal adding U_w.



(d) Implementation of updated path in the commit. New epoch's ratchet tree.

Figure 3.11: Changes of a `Commit` to an MLS ratchet tree. The committer is A_i and the commit contains an `Update`, a `Remove` and an `Add` proposals. The starting ratchet tree is the one presented in Figure 3.10.

The changes to the ratchet key offer FS and PCS with regard to the sender (of the `Update`) member of the group. The changes guarantee PCS because after the `Update` is applied, the key pairs utilized by the member will change, hence, secrets in the next epoch, necessary for the derivation of the epoch secret, will be encrypted to the member with the fresh key pairs. Additionally, the changes provide FS since knowledge of the new encryption key pair will not give access to secrets encrypted to the member in previous epochs.

Remove Proposal

A client $A_i \in \mathfrak{G}_{\mathsf{mls}}^{gid,e}$ of group $\mathcal{G}_{\mathsf{mls}}^{gid}$ can request to remove a client $A_n \in \mathfrak{G}_{\mathsf{mls}}^{gid,e}$ (can be any member of the group) assigned to a leaf of the ratchet tree using a `Remove` proposal. The content of a `Remove` is just A_n's leaf index l to be removed. The processing of a `Remove` (in a `Commit`) results in the following changes to the ratchet tree:

1. Blank the leaf node l specified in the proposal.
2. All intermediate nodes encountered along the path from the removed member's (A_n's) leaf to the root of the tree are blanked.
3. If the root node's right subtree does not have non-blank leaves, then the new ratchet tree's root becomes the left subtree of the old root. Repeat this rule until the right subtree has at least one non-blank leaf. In other words, if the rightmost non-blank leaf has index l', then the ratchet tree has 2^d leaves, where d is the smallest number that satisfies $2^d > l'$.

To illustrate this process, consider the current ratchet tree depicted in Figure 3.11a. The modified ratchet tree after a remove is shown in Figure 3.11b, where in that example, rule three does not result in any changes.

The removal of the leaf node of the member and the erasure of all information from intermediate nodes results in a new ratchet tree for which the removed client A_n possesses no information. This enhances PCS with regard to compromized clients removed from a group.

Add Proposal

A client $B_j \in \mathfrak{G}_{\mathsf{mls}}^{gid,e}$ of group $\mathcal{G}_{\mathsf{mls}}^{gid}$ can request to add a client $U_w \notin \mathfrak{G}_{\mathsf{mls}}^{gid,e}$ to the group using an `Add` proposal. In order to support clients being added to a group while they are offline, a similar concept to pre-key bundles used in Proteus exists in MLS. Clients upload regularly `KeyPackage` information to the distribution server, containing:

- Public Initialization Key (ipk^{U_w}) – This key is used in HPKE to encrypt secrets necessary to initialize the current epoch of the MLS group at

client U_w (to be added). The corresponding private key (ik^{U_w}) is exclusively known to client U_w.

- Leaf node information as specified above.

A copy of a `KeyPackage` of the client being added to a group is the content of an `Add` proposal. The processing of an `Add` proposal (in a `Commit`) results in the following changes to the ratchet tree:

1. If there are blank leaves in the tree, then the index l of the new member is the leftmost empty leaf. Otherwise, the tree is extended by creating a new tree of the same depth d as the current ratchet tree. All nodes in the new tree are blank. Then, the current ratchet tree becomes the left subtree of a new blank node, the root of the new ratchet tree, and the new tree becomes the right subtree. The new ratchet tree has $2^{(d+1)}$ leaves. The index l is the leftmost empty leaf of the new ratchet tree.
2. For all non-blank intermediate nodes encountered along the path from leaf l to the root, add the new member U_w to the node's list of "Unmerged" leaves.
3. The leaf node with index l is assigned the leaf node information from the `KeyPackage` in the `Add` proposal.

To illustrate this process, consider the current ratchet tree depicted in Figure 3.11b. The modified ratchet tree after an `Add` is shown in Figure 3.11c.

Commit Proposed Changes

As already discussed, proposed changes through `Proposal` messages are only applied to a group once a group member receives a `Commit` message sent by another group member. The `Commit` is the last message of an epoch and indicates the beginning of the next epoch. A `Commit` contains a list of `Proposal` messages that members of the group proposed during the epoch and that must take effect (starting from next epoch). The `Proposal` messages must be applied in the order: (1) all `Update` proposals, (2) all `Remove` proposals, (3) all `Add` proposals, as shown in Figure 3.11. Optionally, a `Commit` contains an updated path. The *updated path* is the (non-filtered) path from the leaf node of the member who commits to the root node of the new ratchet tree, but it filters (excludes) all intermediate nodes of which the subtree rooted at their child that is not in the (non-filtered) path does not have non-blank leaves. The updated path is necessary when a commit contains at least one `Update` or `Remove`. The member who commits (the committer) generates the new HPKE encryption key pairs for the aforementioned intermediate nodes on the updated path as follows:

1. All intermediate nodes encountered along the path from the committer's leaf to the root of the tree are blanked.

2. Generate a fresh HPKE key pair for the committer's leaf node.
3. Generate a sequence of *path secrets*, one for each intermediate node in the updated path. `path_secret[0]` refers to the secret of the first node in the path, while `path_secret[r]` where $r >= 0$ refers to the secret of the root node. We do not present the details, but we state that knowledge of `path_secret[i]` implies knowledge of `path_secret[j]` where $j \geq i$.
4. From each `path_secret[i]`, derive an HPKE key pair that replaces the encryption key pair of the i^{th} node in the updated path. (Derivation details omitted).

Furthermore, for each other member of the group (excluding removed clients and newly added clients), the committer will encrypt using the HPKE public encryption keys already available in the ratchet tree before filling the tree with the new path encryption key pairs, the path secret of the first common predecessor of both the committer and the member. For more details on how the committer achieves only logarithmic number of encryptions is described in Section 7.5 of [13]. This encrypted information, alongside the HPKE public key for each node in the updated path, will be part of the `Commit` such that other members can update their view of the ratchet tree. Finally, if the updated path is present, the commit secret for the next epoch is the `path_secret[r+1]`, that is, the secret derived from the path secret of the root, which is known to everyone. Otherwise, the commit secret is the all-zero byte string.

Regarding added members, each will learn its corresponding path secret through a `Welcome` message, which also contains a secret available to only existing members of the group from the previous epoch, required to derive the next epoch secret. This information of a `Welcome` is encrypted to the added client using the HPKE public initialization key available in the `KeyPackage` of the `Add` proposal that added the client to the group. As opposed to `Handshake` and `Application` a `Welcome` message is sent exclusively to the client being added and not the group.

We highlight that when an updated path exists in a `Commit`, the committer gains the advantages of PCS since an adversary loses access to possibly corrupted key pairs known at the committer. Figure 3.11d, shows the modifications to the ratchet tree, as a result of the updated path.

3.7.3 Wire MLS Decisions

The MLS standard leaves many decisions open to the application designer. In this section, we list the choices made by Wire that are most relevant to our analysis.

Orchestrator Delivery Service

`Handshake` messages propose and commit changes on a specific epoch's ratchet tree. All `Handshake` messages in Wire MLS are sent by default as `PublicMessage`, meaning they are not encrypted but only authenticated. The reasoning behind this decision is that the distribution server S_{DS} orchestrates clients belonging to a group, meaning S_{DS} decides which `Commit` and `Proposal` are accepted and routes them to clients. Therefore, knowledge of the content of `Handshake` messages is essential for maintaining the most recent view of groups' memberships at the most recent epochs. Additionally, when the server receives a `Proposal` or a `Commit` from a group member, authorization checks based on application-specific rules can be performed to decide whether the member is allowed to execute these actions. If the server returns an error indicating not authorized activity to the client, then the `Proposal` or `Commit` is removed as if it had never happened.

Moreover, the orchestrator server S_{DS} is responsible for resolving conflicting `Commit` messages for the same epoch. Due to the asynchronous nature of the messaging system, two or more group members may simultaneously commit changes on the same epoch. However, `Commit` messages from different members will result in diverse MLS group states for the next epoch (if some of the members follow one and the rest the other). Therefore, it is required that all members of a group agree on the same `Commit` message either by preventing conflicting messages from occurring or by developing tiebreaking rules for deciding which is the definite `Commit`. The orchestrator server S_{DS} ensures that only one `Commit` is delivered to the members of the group for each epoch. In case many authorized members attempt to commit, only one of them will receive a successful response, while the rest are informed about the conflict and that they have to wait for the selected `Commit` to be delivered to them.

Out-of-Order Application Messages

Every `Application` message is enclosed in a `PrivateMessage` encrypted using keys derived from the application symmetric ratchets in the current epoch's secret tree. However, due to the asynchronicity of message delivery, a `PrivateMessage` encrypted using keying material from an older epoch can be delivered to other group members when they are already in a later epoch. Consequently, members of the group who do not maintain secrets and the state of groups in previous epochs cannot decrypt them. Wire clients are configured to store the last three epochs' states of groups in addition to the current epoch state to enable out-of-order `Application` message decryption.

Credentials

As already discussed in the Proteus E2E protocol, verifying the identity of the other communicating parties out-of-band is essential; otherwise, the distribution server S_{DS} can trivially perform impersonation attacks. The same security considerations apply to MLS and the `KeyPackage` that the server maintains and sends to other clients to enable the addition of members to groups. Each `KeyPackage` and leaf of the ratchet tree contain credential information and a public signature key.

In Wire MLS, the credentials are planned to be x509 certificates issued by the ACME server S_{ACME}. The clients follow the standard ACME procedure for obtaining a certificate from the ACME server as described by the ACME standard [15]. Wire defines some custom challenges in their E2E identity solution[9], not part of the ACME standard, that the client has to fulfill to get authorization for obtaining a certificate for a specific identity. We omit details of the custom challenges as they are not important for our analysis. These certificates will become the credentials used in `KeyPackage` and part of the leaf nodes of the ratchet tree of a group when the creator of the `KeyPackage` becomes a member of the group. Importantly, the root of trust is the ACME server, and clients can automatically verify the identity of other clients because they trust the CA at the ACME server. The MLS group's status will then be automatically determined by verifying the certificates presented in the credentials. If all members of an MLS group have valid credential then the group is automatically verified. Thus, out-of-band verification will no longer be required.

[9] https://github.com/wireapp/rusty-jwt-tools/blob/main/e2e-identity/README.md

Chapter 4

Analysis of Wire - Proteus

In this chapter, we dive into the analysis of the Proteus E2E protocol which we extensively described in Section 3.6. We first present the threat model we are considering in Section 4.1. Then, in Section 4.2, we provide a high-level discussion of the vulnerabilities, followed by detailed descriptions of the attacks and the proposed mitigations. For each attack presented in this chapter, we propose a mitigation in order to limit or completely alleviate its impact. Moreover, we confirmed all attacks on Android legacy version 3.82.38 and Android new version 4.2.4 applications. For this task, we implemented a malicious distribution server capable of performing the attacks, instructed through a web interface, to which the clients connected instead of Wire's production distribution servers.

4.1 Threat Model

Wire does not directly list the security goals they aim for nor the threat model they are considering. Due to its nature of being an E2E messaging system used by high-risk organizations, the following is a realistic threat model that we consider throughout our analysis:

- A malicious actor M who controls the Wire distribution server S_{DS}, gaining the power of a mediator between group communications and read-write access to the server's contents and data structures. The malicious actor can reorder and drop messages in a Proteus dialogue, send new arbitrary messages to a client (this is not a capability to send encrypted messages), and bypass the high-level API user authentication mechanisms. We refer to this model as the *Malicious Wire Server*.

Our analysis does not consider network adversaries since we view the usage of TLS as a secure solution for the C2S protocol.

4.2 High-Level Description of the Vulnerabilities

The vulnerabilities found in Wire Proteus E2E communication, exploited in our attacks, revolve around two mistakes.

- First, as studied by Cremers et al. for Signal in [30], improper session handling can severely weaken PCS guarantees for a Double Ratchet implementation. The realization of dialogues between clients as a session management protocol such as the Proteus dialogues, where multiple parallel sessions opaque to clients can coexist, can cause several issues. Although the underlying Double Ratchet protocol provides FS and strong-PCS, more is needed to prove that the dialogues as a whole maintain the same level of security. Indeed, in the following attacks, we show that Proteus dialogues never heal after corruption.
- Second, the absence of cryptographic means for securing the authenticity and integrity of information relayed through or maintained at the distribution server and the ultimate trust given to the server can result in many unwanted outcomes. An example is the client's independence from the other clients of a user and the lack of signatures on users' client lists, leading to trivial confidentiality violations. Another example is the optional signature in pre-key bundles; its optionality degrades FS. Last, metadata not protected by MAC tags: manipulation can result in displaying messages in an improper way, for example out-of-order or in the wrong conversation.

4.3 Proteus Dialogue Attacks

In Section 3.6, we thoroughly described how E2E channels between clients are realized in the context of Proteus, using Proteus dialogues. In this section, we illustrate a family of attacks exploiting the weaknesses of Proteus dialogues, a consequence of the improper and opaque to clients session management. In contrast to Cremers et al.'s study on Signal's Sesame session management, Wire's session management through Proteus dialogues completely voids PCS guarantees, even in a weak adversary model with limited compromise capabilities (weak-PCS), unless clients manually reset the affected dialogues. The main difference is that new sessions are initialized only during dialogue initialization while in Sesame this is done automatically every one hour which eventually leads to eviction of the malicious session. For the PCS violation attacks of Sections 4.3.2, 4.3.3 and 4.3.4, we can reasonably assume that a manual reset never occurs. This assumption is based on the absence of decryption errors or any other errors in the conversation's application interface, which would typically trigger the user's suspicion that something is wrong. We also see that, in the presence of a

strong adversary and even if manual resets occur, PCS guarantees are still violated.

In what follows, we consider the Malicious Wire Server threat model as a starting point and gradually increase the adversary's capabilities. Each attack explicitly specifies the assumptions made. Moreover, for simplicity, in this section we assume that each user has a single client and as an abuse of notation we omit the subscript client identifier thus conflating the user identifier with the client identifier, e.g., for Alice, A also denotes her client.

4.3.1 Attack 1 (Trivial Message Replay)

Let A and B be communicating clients with Proteus dialogue $\mathcal{D}_B^A$ and $\mathcal{D}_A^B$, respectively. Let $ps_{A\leftrightarrow B}^{tag}$ be a Proteus session managed by the dialogue between them. We show a flaw that allows the malicious actor M to replay in Proteus dialogue $\mathcal{D}_A^B$ all the messages which were encrypted using the first sending symmetric ratchet of client A originating from the chain key $ck_{1,1}^A$ of any Proteus session $ps_{A\leftrightarrow B}^{tag}$ (managed by the dialogue) between A and B where A is the initiator. A requirement for this attack to be successful is that A used the last resort bundle of B during the XPDH key exchange, which contains a short-term key pair that multiple session initializations can reuse. We exploit this fact to employ our attack.

Attack Description and Impact

Setting Let A and B be communicating clients and assume a Proteus dialogue $\mathcal{D}_B^A$ at A managing a single Proteus session $ps_{A\rightarrow B}^{tag}$, where A is the initiator, and the pairing dialogue $\mathcal{D}_A^B$ at B managing a single Proteus session $ps_{A\leftarrow B}^{tag}$, where B is the responder. Furthermore, assume that the Proteus session was initialized with the last resort bundle of B obtained from the server. We emphasize that the malicious actor M controlling the server can deliberately provide the last resort bundle on every client's request for a bundle.

Description With these dialogues in place, the two clients, A and B, can send E2E encrypted messages to each other through the adversary-controlled server. Although adversary M cannot read these messages, M can collect them. The attacker's goal is to force B's Proteus dialogue $\mathcal{D}_A^B$ to accept replayed `PreKeyMessages` that A sent to B, using $ps_{A\rightarrow B}^{tag}$ through $\mathcal{D}_B^A$, by initializing a new Proteus session $ps_{A\leftarrow B}^{tag'}$ in $\mathcal{D}_A^B$ with the same session tag ($tag' = tag$). Note that the attacker M cannot simply replay a message because of the Proteus sessions' duplication detection (each key is used only once). The goal is achievable only when A used the last resort bundle to initialize $ps_{A\rightarrow B}^{tag}$ because many session initializations can reuse the same

bundle. Furthermore, since a `PreKeyMessage` carries the information for the pre-key bundle identifier, and the public long-term identity and base ephemeral keys in the associated data then the XPDH performed at the responder B will result into the same master secret in the new Proteus session $ps^{tag'}_{A\leftarrow B}$. Last, the first public ratchet key of the initiator is also part of the associated data of the `PreKeyMessage`. Consequently, the same chain key $ck^{A}_{1,1}$ is derived, which leads to the first sending symmetric ratchet of A to be the same, meaning that all the message keys used for `PreKeyMessages` of $ps^{tag}_{A\leftarrow B}$ are the same in $ps^{tag'}_{A\leftarrow B}$. Hence, all these `PreKeyMessage` can be replayed in the new Proteus session. We highlight that the MAC tag in a replayed `PreKeyMessage` is correct since the adversary didn't modify the messages and that the adverary cannot send new messages since M is unable to derive the master secret. Below, we explicitly specify the conditions under which the attacker can replay `PreKeyMessages`.

Replay Limitations A subtle restriction mandates that $ps^{tag}_{A\leftarrow B}$ managed by $\mathcal{D}^{B}_{A}$ at B advanced at least to the sixth receiving symmetric ratchet before replaying the messages. Therefore, M has to wait until it observes from the associated data of encrypted messages that there were enough asymmetric steps by observing the public ratchet keys and the session tag. In more detail, according to the Proteus dialogue decryption algorithm described in Section 3.6.3, when a `PreKeyMessage` c is decrypted, it can potentially lead to the initialization of a new Proteus session either when no session in $\mathcal{D}^{B}_{A}$ has the tag in the associated data of the `PreKeyMessage` or when an `InvalidSignature` error is encountered. However, a Proteus session always retains the five previous receiving ratchets. Thus, if the first receiving ratchet of B in $ps^{tag}_{A\leftarrow B}$ is still present (the first sending ratchet of A) and a `PreKeyMessage` is replayed, it will trigger an error different than the `InvalidSignature` caused by a MAC tag failure (most probably a `Duplicate` error). The `InvalidSignature` error is necessary for the Proteus dialogue decryption algorithm to initialize a new Proteus session $ps^{tag'}_{A\leftarrow B}$ (and replace the previous), with $tag' = tag$ when it already manages a session with the same tag (see Section 3.6.3 for details). Hence, the new Proteus session $ps^{tag'}_{A\leftarrow B}$ which is now managed by $\mathcal{D}^{B}_{A}$ becomes the active session $\mathcal{D}^{B}_{A}.\mathsf{active} = ps^{tag'}_{A\leftarrow B}$ and $\mathcal{D}^{B}_{A}$ will successfully decrypt the `PreKeyMessage`, as long as the same last resort key pair is still in use by B.

After obtaining the decrypted/replayed message $m \leftarrow \mathcal{D}^{B}_{A}.dec(c)$, it is forwarded to the application which decides how to display it according to its type. Recall from Section 3.5, that every message has a message identifier. Since the message is replayed, the identifier will be the same. The two Android applications demonstrated different behaviors.

- In the legacy Android application, we couldn't observe anything in the conversation's interface by replaying messages (of any type) because the application checks the message identifier of previously received messages and ignores duplicates.
- Replaying a `PreKeyMessage` succesfully displayed `Ephemeral` and `Text` messages on the new Android application conversations. For the latter type, the message was displayed if and only if the message with that particular identifier was edited or deleted. This happens because the behavior of the new application changed compared to the legacy version and the identifiers are not considered when the previous version of the message is removed from the conversation.

Notice that M cannot replay messages for the second sending ratchet of A because the randomly chosen ratchet key pair of B will differ, with high probability, from the corresponding ratchet key pair chosen the first time. An additional observation is that, since the replayed `PreKeyMessage` had the same tag as the previous time they were sent, then $ps^{tag}_{A\leftarrow B}$ will be overwritten in $\mathcal{D}^B_A$ by the new Proteus session state $ps^{tag'}_{A\leftarrow B}$. Consequently, client A can no longer send encrypted messages using $ps^{tag}_{A\rightarrow B}$ which is currently active for dialogue $\mathcal{D}^A_B$. Similarly, neither can B send encrypted messages to A. If any such message is delivered then a decryption failure will be detected on the receiving side. Users will eventually realize that something has gone wrong and to resolve the issue at least one of the Proteus dialogues should be reset, described in Section 3.6.3. It is essential to highlight that M cannot read the content of the replayed messages, but only replay old messages.

Mitigations

The replay attack can be completely mitigated if the last resort bundles are discontinued at the cost that when the server exhausts all pre-key bundles, offline initialization of sessions is no longer supported. However, there are two suggestions for improving security against such an attack while last resort bundles are kept.

First, we suggest that the behavior of the legacy Android application is implemented across all supported Wire applications. This includes the new Android application, which turned out to accept replayed messages. Previous message identifiers must be stored in a database, and when a new message is decrypted and forwarded to the application, the application must check against this database for duplicates. A message is rejected when it is marked as duplicate. The intuition behind this suggestion is that the adversary cannot change the message identifier in a replayed message since it is encrypted and authenticated. Additionally, in order to prevent storing all message identifiers for the entire lifetime of a client, we can safely reset

the database responsible for duplicate checks after the last resort bundle is updated since the problem persists as longs as the same last resort key pair is in use by a client.

Second, we suggest a change in the Proteus dialogue decryption algorithm presented in Section 3.6.3. We noticed that when a Proteus dialogue decrypts a `PreKeyMessage` with a session tag in its associated data of a session already managed by the dialogue, and if an `InvalidSignature` error occurs, the dialogue gives a second chance and attempts to initialize a new session which replaces the previous existing one, if initialization is successful. In order to avoid having messages replayed in Proteus dialogues that already manage a session with the same tag as the one contained in the replayed message, we propose to remove this special treatment of the `InvalidSignature` error. Instead, the Proteus dialogue must treat the error as a decryption failure, meaning that a new Proteus session will not be created to decrypt the replayed `PreKeyMessage`. This suggestion protects a Proteus dialogue from a replay as long as the dialogue still manages the targeted Proteus session.

4.3.2 Attack 2 (Disguised Mallory 1)

For the rest of the attacks in this section, we present more powerful attacks requiring additional adversarial capabilities. In this attack, we show how the malicious actor M can violate the PCS guarantees of a Proteus dialogue $\mathcal{D}_A^B$ that B uses to communicate with A. The idea of the attack is that the attacker M, by exploiting session management, tricks B into exchanging messages with M, while B believes that it exchanges messages with A, because Proteus dialogue $\mathcal{D}_A^B$ is still employed for encryption and decryption.

Adversarial capabilities We consider two possible adversarial models under which the attack is analyzed.

- A weak adversary M can temporarily perform (via an oracle) DH computations using the private long-term identity key of one of the communicating clients without revealing it (limited corruption).
- A strong adversary M can reveal the private long-term identity key of one of the communicating clients (full corruption).

We present the attack in the context of the weak adversary model in order to highlight some important aspects, noting that the strong adversary model attack is identical. The reason for distinguishing between the adversaries is that we propose mitigations for both models, with the mitigations for the weak adversary being less invasive to the current philosophy of Proteus dialogues.

Attack Description and Impact

Setting We start the attack description by assuming a Proteus dialogue $\mathcal{D}_B^A$ at A managing a single Proteus session $ps_{A\to B}^{tag}$, and the pairing dialogue $\mathcal{D}_A^B$ at B managing a single Proteus session $ps_{A\leftarrow B}^{tag}$, as shown in Figure 4.1a. Without loss of generality, we assume that A is the initiator and B is the responder for the single session managed by the Proteus dialogues. Furthermore, it is irrelevant to the attack for how long the two parties were communicating.

Description After M obtains temporary access to an oracle that performs DH computations using the private long-term identity key idk^A of A (Figure 4.1b), the attack proceeds as follows. First, M initializes a Proteus session $ps_{M\to B}^{tag'}$ from a pre-key bundle of B, as described in Section 3.6.2, with the difference that the computation of the $(prepk_x^B)^{idk^A}$ DH share is done using the temporary access to the corruption oracle. We highlight that M is not restricted to the cryptographic libraries provided by Wire; consequently, M can alter the algorithms and directly access Proteus sessions without a wrapper Proteus dialogue. Now, M can use directly $ps_{M\to B}^{tag'}$, where $tag' \neq tag$, to encrypt a message m, resulting in $c \leftarrow ps_{M\to B}^{tag'}.enc(m)$, and send c to B with A's sender information. When B receives c, it attempts to decrypt it using dialogue $\mathcal{D}_A^B$. However, since c is a `PreKeyMessage` (since it is the first message sent by the initiator in the Proteus session) with a session tag different than any other session managed by $\mathcal{D}_A^B$, it will result in the initialization of a new session $ps_{M\leftarrow B}^{tag'}$. Additionally, this session becomes the active session for $\mathcal{D}_A^B$ and coexists with $ps_{A\leftarrow B}^{tag}$ in $\mathcal{D}_A^B$. Therefore, any subsequent messages B encrypts using $\mathcal{D}_A^B$ are "destined" to M; hence, special care on message delivery is required so that A does not notice the attack. The adversary can easily prevent this by observing the session tags in the associated data of encrypted messages. The current state of the Proteus dialogues is shown in Figure 4.1c.

Notice that when A sends a message to B using $\mathcal{D}_B^A$, session $ps_{A\to B}^{tag}$ is employed. This can reactivate the original communication session $ps_{A\leftarrow B}^{tag}$ mamaged by $\mathcal{D}_A^B$ at B, as depicted in Figure 4.1d. Most interestingly, when M loses access to the corruption oracle, it can still use session $ps_{M\to B}^{tag'}$ to exchange messages with B, while disguised as A (Figure 4.1e). Additionally, M can precompute before losing access to the corruption oracle, the DH shares $(lrpk^B)^{idk^A}$ and $(prepk_x^B)^{idk^A}$ for all ephemeral pre-key bundles that B already uploaded to the server. Consequently, M can perform the attack even after losing access to the corruption oracle, and as long as B uses the same short-term key pair or the ephemeral pre-key bundles of B available

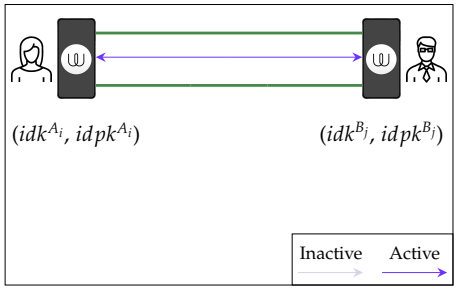
during corruption are not employed in session initializations at B.

Last, we want to stress that adversary M can perform the attack even against client A when corrupting A because clients can have a Proteus dialogue with themselves. Therefore, in a conversation where only users A and B are participating, we can make both users A and B have the same conversation transcript; hence, the attack is not detectable by comparing the messages shown in a conversation's application interface. This attack generalizes to more than two users.

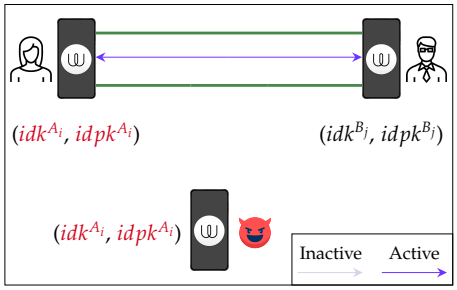
Impact The attack is highly robust since the initial communicating clients A and B can still communicate using the session managed by their Proteus dialogues before the intervention of M. We emphasize that M cannot read encrypted messages that A sends to B nor messages that A can decrypt (sent by B). The attack highlights a severe weakness in the session management employed by Wire, which allows constant switching between sessions unbeknownst to B. The Proteus dialogue internal changes from one Proteus session to another are entirely opaque to the Wire client application; therefore, client B cannot notice the transitions. As a naive attempt to notify the receiving client about new Proteus session initializations in a Proteus dialogue, Wire's applications currently use the `ResetClientAction` special message type, described in Section 3.5. However, an attacker M is not forced to send that type of message. Most importantly, dialogue $\mathcal{D}_A^B$ at B used for communication with A never heals as promised by PCS because M has the potential to continue adversarial behavior any time in the future with an already established malicious session that $\mathcal{D}_A^B$ never evicts under the assumption of no manual resets.

Fine-Grained Reactivation of Communication Between A and B Additionally, we show that attackers can withdraw from their position at will and restore the E2E communication between A and B without them noticing, therefore completely shattering the robustness of PCS (as defined in Cohn-Gordon et al. [26]). In general, B will encrypt messages with the currently active Proteus session of dialogue $\mathcal{D}_A^B$, the last session used to decrypt a message. Instead of forwarding every message to its destination, M can devise a different strategy so that an encrypted message always exist that M can later use to reactivate the original communication session at B. The idea is that before reactivating or creating for the first time the malicious session, M withholds one message A sends to B using the original session. Then, whenever A sends another message using the original session to B, M can safely forward the previously withheld message and retain the newly received one. Discerning the session from which a message comes from can be done by looking at the associated data of the message. A more invasive solution can delay all messages from A. This strategy gives more control

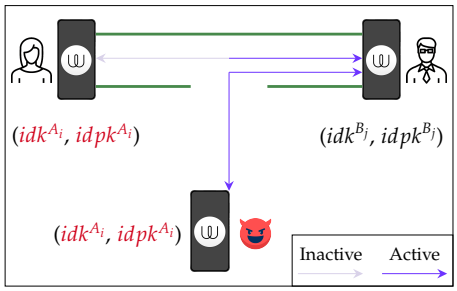
to adversary M since it decides when to restore the original E2E communication. We refer to this improvement that transforms at the will of the adversary the Proteus dialogues' states to that in Figure 4.1d as the *reactivation trick* and reuse it in the following attacks.



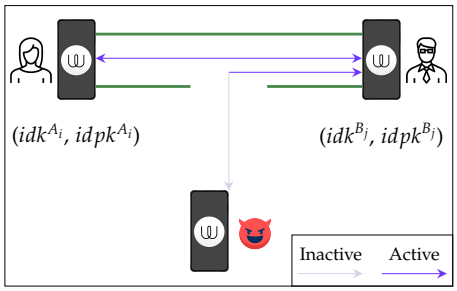
(a) Initial Proteus dialogue between clients of Alice and Bob, before corruption.



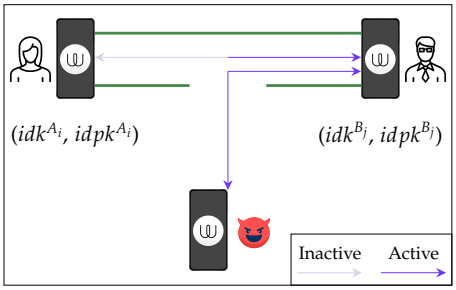
(b) After limited corruption of Alice client's long-term identity key pair.



(c) After initialization of malicious Proteus session.



(d) After reactivation trick. Also assume Mallory loses access to corrupted private keys.



(e) After usage of existing inactive malicious Proteus session.

Figure 4.1: Proteus dialogue's state between clients of Alice (A) and Bob (B) during the Disguised Mallory 1 attack (without loss of generality, we present the attack when Alice's client is corrupted). With green color we represent the Proteus dialogue. The Proteus sessions it manages are colored purple. The direction of Proteus sessions arrows is interpreted as follows: the client's Proteus dialogue that handles the Proteus session is the opposite to where the arrow head ends up.

Mitigations (Weak Adversary)

In the weak adversary model, we suggest three improvements in order to achieve three goals:

1. We want to make the reactivation trick and, in general, reactivation of old sessions in a Proteus dialogue infeasible so that an adversary can no longer reactivate malicious sessions.
2. We want to fix the new problems introduced by fulfilling the first goal. In particular, the first fix breaks the solution for simultaneous Proteus dialogue initializations at both communicating parties. Previously, when two clients were initializing simultaneously their Proteus dialogues for communication with each other, there were two distinct Proteus sessions; therefore, they had to agree on one. Session reactivation on successful decryption was the solution to this challenge because eventually, both clients in a dialogue will use the same underlying session to encrypt a message when no other message reactivating another session is yet to be delivered to them.
3. We want to eventually evict the malicious session from the victim's Proteus dialogue so that the dialogue heals and PCS is achieved.

We now list our suggestions to address all the goals listed above:

1. In order to remove the reactivation trick from the equation, we propose a modification to the Proteus dialogue decryption algorithm. In particular, we adopt the solution proposed by Cremers et al. in [30] for Signal's Sesame session management. The solution advises that the active session must be the most recently initialized, and reactivation of existing sessions managed by a Proteus dialogue is forbidden. Note that decryption using previous sessions will still return the decrypted message without reactivating the session.
2. Sessions managed by a Proteus dialogue, after applying our first suggestion, cannot be reactivated once a new session is initialized. This causes problems, in particular, with simultaneous dialogue initializations from pre-key bundles. As an example, assume that both A and B initialized their dialogues $\mathcal{D}_B^A$ and $\mathcal{D}_A^B$ simultaneously. $\mathcal{D}_B^A$ at A will have a single session $ps_{A\to B}^{tag}$, while $\mathcal{D}_A^B$ at B will have a different session $ps_{B\to A}^{tag'}$. Suppose both of them simultaneously encrypt two messages using $\mathcal{D}_B^A$ and $\mathcal{D}_A^B$, resulting in $c_A \leftarrow \mathcal{D}_B^A.enc(m_A)$ and $c_B \leftarrow \mathcal{D}_A^B.enc(m_B)$, respectively. Then, the encrypted messages are simultaneously delivered to the recipients. In that case, both $\mathcal{D}_B^A$ and $\mathcal{D}_A^B$ will initialize and switch to the other session (because both c_A and c_B are `PreKeyMessage`). Communicating using two different sessions will result in losing PCS because no asymmetric steps will be performed.

To resolve this problem, we propose that Proteus dialogues automatically initialize a new session when both:

a) the session is not active in the Proteus dialogue.

b) and exactly a constant number of messages were successfully decrypted using that session after it became inactive.

Consequently, only a bounded number (specified by a constant) of messages will be affected and then a new session initialization will be triggered, eventually leading to consensus on the session that the Proteus dialogues will use. Note that we allow a non-active session to decrypt messages as long as it is managed by the Proteus dialogue.

3. Finally, in order to eventually evict a malicious session (recall that sessions are removed in a FIFO fashion) from a Proteus dialogue, we suggest that a Proteus dialogue initializes a new session automatically every hour, which is similar to Signal's Sesame session management. Note that as long as a Proteus dialogue manages a malicious session, the adversary can use it to send encrypted messages, however, the first suggestion prevents its reactivation on the recipients side, meaning that the victim will no longer send messages to the adversary. Furthermore, implementing this proposal is necessary (assuming fix one is implemented) to recover from the attack of Section 4.3.4. Notice that either this suggestion or the previous one will first initialize a new session, with each being more appropriate in different scenarios. For instance, the previous suggestion restricts the number of affected messages on simultaneous initializations instead of waiting one hour where the number of messages exchanged is unbounded.

Recall from the attack decription that the adversary M can precompute DH shares when it has access to the corruption oracle. In Section 4.4.5, we uncover that clients do not update the short-term last resort key pair. Consequently, weak-PCS is not achieved even if the above countermeasures are implemented, until Wire's clients start updating last resort key pairs.

The above-presented mitigations also apply to the attacks of Sections 4.3.3 and 4.3.4.

Mitigations (Strong Adversary)

In the face of a strong adversary, the mitigations against a weak adversary do not recover PCS because a strong adversary can initialize new sessions in a Proteus dialogue any time in the future. Therefore, there is no hope when a session management solution is employed. However, at the current state of Wire, Proteus dialogues are useful only for scenarios where simultaneous initializations occur. Hence, no real incentive exists for utilizing such a solution just to resolve concurrent initializations at the cost of losing PCS against

a strong adversary. We suggest that the Proteus dialogue session management solution is abandoned and replaced by a single Proteus session with a tiebreaker rule. Our suggestion is more invasive to the current philosophy of Proteus dialogues and would require more changes.

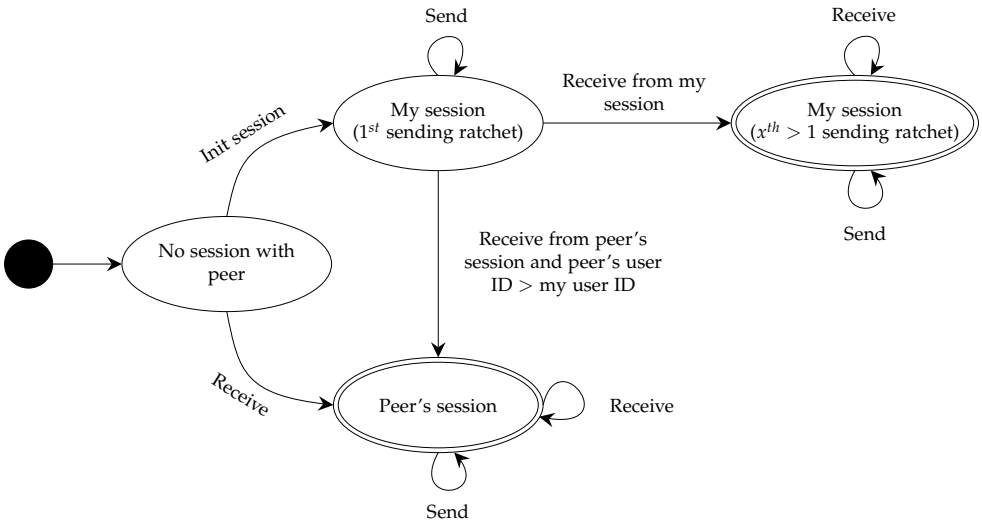


Figure 4.2: State machine for the proposed dialogue between clients using a single Proteus session with a tiebreaker. When a message is received and decryption fails the transition is not taken.

Let's consider client A of Alice with user identifier ID_A and B of Bob with user identifier ID_B are about to establish a dialogue. If no simultaneous initialization of Proteus sessions is taking place, then consensus is reached without a tiebreaker. In particular, when one of the clients initializes a Proteus session using a pre-key bundle of its peer and sends an encrypted message, which is delivered to the peer before it attempts to initialize a Proteus session, the peer accepts and marks as definitive the session in the dialogue. Otherwise, if simultaneous initializations occur, then consensus is reached using a tiebreaker. In more detail, when both clients simultaneously initialize a Proteus session using a pre-key bundle of their peers, an agreement must be reached. The proposed tiebreaker rule instructs the client whose user has the smaller identifier to adopt its peer Proteus session and mark it as definitive in the dialogue. For instance, when $ID_A > ID_B$, client B will adopt the Proteus session initialized by A in the case of concurrent initialization, according to the tiebreaker. Notice that B might have sent multiple messages before noticing the simultaneous initialization. In such case, B is required to re-encrypt them using the definitive Proteus session in the dialogue. Therefore, all dialogues should maintain a buffer of all messages sent using a Proteus session before it is marked definitive so that they can re-encrypt them later using the final Proteus session (if it changes). Figure 4.2

shows the state machine for the proposed dialogue using a single session.

The above-presented mitigations also apply to the attacks of Sections 4.3.3 and 4.3.4.

4.3.3 Attack 3 (Disguised Mallory 2)

This attack shows how the malicious actor M can violate the PCS guarantees of a Proteus dialogue $\mathcal{D}_B^A$ that A uses to communicate with B by using the short-term last resort key of A. The attacker M, by exploiting session management, tricks A into exchanging messages with M, while A believes that it exchanges messages with B because Proteus dialogue $\mathcal{D}_B^A$ is still employed for encryption and decryption. The attack in this section is similar to the attack in Section 4.3.2, with small differences. First, the adversarial capabilities differ from what was previously presented: the adversary corrupts a short-term last resort key instead of a long-term identity key. Second, this attack has characteristics of a Key Compromise Impersonation (KCI) attack, where corrupting one party (A) allows for the impersonation of arbitrary parties (B) to the corrupted party (A). Last, the derivation of the malicious session's master secret that M uses to perform the attack slightly differs from what is presented in Section 3.6.2.

Adversarial capabilities We consider two possible adversarial models under which the attack is analyzed.

- A weak adversary M can temporarily perform (via an oracle) DH computations using the private short-term last resort key of one of the communicating clients without revealing it (limited corruption).
- A strong adversary M can reveal the private short-term last resort key of one of the communicating clients (full corruption).

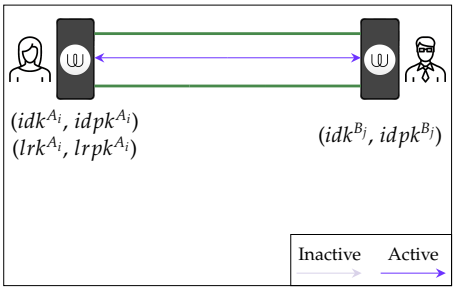
We present the attack in the context of the weak adversary model.

Attack Description and Impact

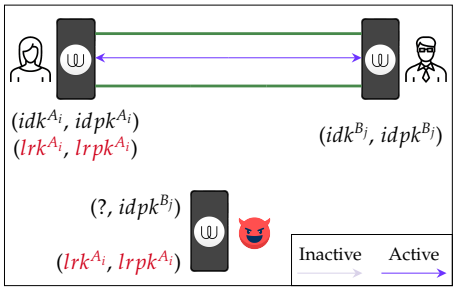
The attack proceeds exactly as in Section 4.3.2, with a small difference in deriving the malicious session's master secret. Thus, all observations, remarks, comments, and mitigations presented in that section also apply to the attack under discussion. In order to clear all ambiguities, we present the details of the master secret derivation.

After M obtains temporary access to an oracle that performs DH computations using the private short-term last resort key lrk^A of A (Figure 4.3b), the master secret is derived as follows:

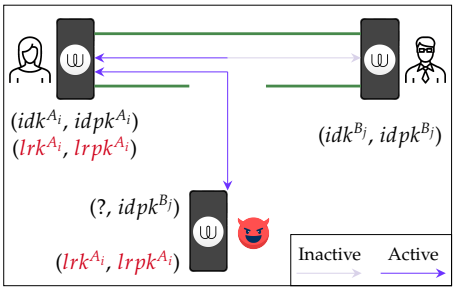
$$ms \leftarrow (idpk^B)^{lrk^A} \| (idpk^A)^{ebk^M} \| (lrk^A)^{ebk^M}$$



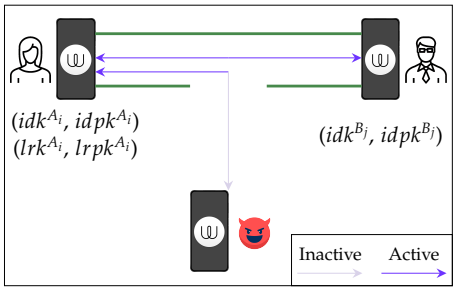
(a) Initial Proteus dialogue between clients of Alice and Bob, before corruption.



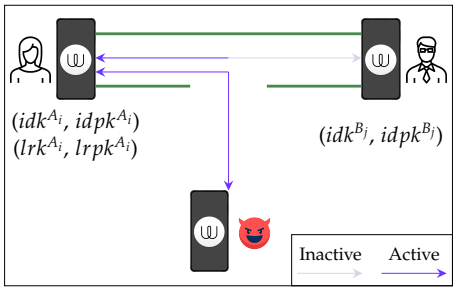
(b) After limited corruption of Alice client's short-term last resort key pair.



(c) After initialization of malicious Proteus session.



(d) After reactivation trick. Also assume Mallory loses access to corrupted private keys.



(e) After usage of existing inactive malicious Proteus session.

Figure 4.3: Proteus dialogue's state between clients of Alice (A) and Bob (B) during the Disguised Mallory 2 attack (without loss of generality, we present the attack when Alice's client is corrupted). With green color we represent the Proteus dialogue. The Proteus sessions it manages are colored purple. The direction of Proteus sessions arrows is interpreted as follows: the client's Proteus dialogue that owns the Proteus session is the opposite to where the arrow head ends up. Notice the differences to the malicious session compared to Figure 4.1. This attack has characteristics of a Key Compromise Impersonation (KCI) attack.

The computation of the $(idpk^B)^{lrk^A}$ DH share is done using the temporary access to the corruption oracle. Note that M can precompute this DH share and use it after losing access to the corruption oracle. The DH share is equivalent to the DH value $(lrpk^A)^{idk^B}$ B would have computed for initializing a session with A. Furthermore, notice that $idpk^B$ is present in every pre-key bundle B uploads to the server; thus, it is available to M, and the ephemeral

base key pair is of M's choice.

The Proteus dialogue states as the attack proceeds are illustrated in Figure 4.3. We highlight that corrupting A allows for compromising the security of Proteus dialogues A has with other clients.

4.3.4 Attack 4 (Mallory-in-the-Middle)

In this attack, we combine the previous two attacks in a way such that M can read, modify, and insert messages in the Proteus dialogue that clients A and B use to communicate. Our goal as M will be to establish two Proteus sessions, one disguised as A and the other as B. The rest of this section covers the combination that assumes an adversary who corrupts the smaller number of communication parties. This attack can clearly be executed if the adversary manages to execute exactly one of the Disguised Mallory 1 or 2 attacks against both A and B. However, we will instead combine the two attacks to obtain a MitM position while only compromising a single client.

Adversarial capabilities We consider two possible adversarial models under which the attack is analyzed.

- A weak adversary M can temporarily perform (via two oracles) DH computations using the private long-term and short-term last resort keys of one of the communicating clients without revealing them (limited corruption).
- A strong adversary M can reveal the private long-term and short-term last resort keys of one of the communicating clients (full corruption).

We present the attack in the context of the weak adversary model.

Attack Description and Impact

We now present a generic attack which can be instantiated using an adversary with different capabilities and combinations of the Disguised Mallory attacks. In our context, the instantiation is done by respecting our previous assumptions about the adversary's capabilities.

We assume a Proteus dialogue $\mathcal{D}_B^A$ at A managing a single Proteus session $ps_{A\to B}^{tag}$, and the pairing Proteus dialogue $\mathcal{D}_A^B$ at B managing a single Proteus session $ps_{A\leftarrow B}^{tag}$, as shown in Figure 4.4a. Without loss of generality, we assume that A is the initiator and B is the responder for the single session managed by the Proteus dialogues. Furthermore, it is irrelevant to the attack for how long the two parties were communicating.

The adversary's goal is to obtain two Proteus sessions, $ps_{M\to A}^{tag'}$ and $ps_{M\to B}^{tag''}$, such that $ps_{M\to A}^{tag'}$ (resp. $ps_{M\to B}^{tag''}$) can be used to send encrypted messages to

A (resp. B) disguised as B (resp. A) and A (resp. B) will use its Proteus dialogue $\mathcal{D}_B^A$ (resp. $\mathcal{D}_A^B$) to decrypt them by internally using $ps_{M\leftarrow A}^{tag'}$ (resp. $ps_{M\leftarrow B}^{tag''}$). By fulfilling this requirement, M can now decrypt all messages, read, even modify them or add new messages and then encrypt and send them to the other client's Proteus dialogue, as a full MitM.

Moreover, we want to be able to perform the reactivation trick in order to withdraw M from the MitM position. Hence, the Proteus session $ps_{A\rightarrow B}^{tag}$, where $tag \neq tag'$ (resp. $ps_{A\leftarrow B}^{tag}$, where $tag \neq tag''$), must still be managed by dialogue $\mathcal{D}_B^A$ (resp. $\mathcal{D}_A^B$).

The obvious instantiation that satisfies the requirements is to employ Disguised Mallory 1 to impersonate A to B and Disguised Mallory 2 to impersonate B to A, as shown in Figure 4.4.

4.4 Other Attacks and Findings

In this section, we mainly investigate attacks related to the absence of cryptographic means for authenticating and integrity protecting information relayed through the distribution server S_{DS}, which inadvertently results in needing to put full trust in the server. Moreover, we discuss some interesting findings about the number of asymmetric ratchet steps required in a Proteus session to heal after a full state corruption. Last, we discuss the findings of the Clone Attack experiment in [28].

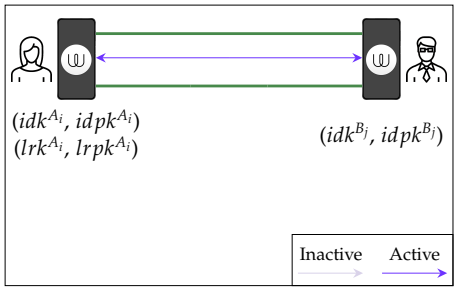
For the attacks in this section, we consider the Malicious Wire Server threat model without any additional assumptions regarding the capabilities of the adversary M.

4.4.1 Attack 5 (Trivial Confidentiality Violation)

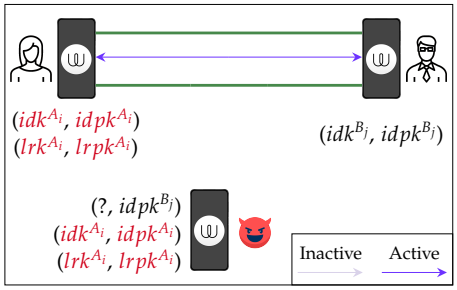
In Section 3.3.2, we studied the procedure users follow to register clients under their control. In essence, there is no mechanism for cryptographically binding clients of a user together. Additionally, as discussed in Sections 3.3.2 and 3.6.4, a client readily accepts the list of clients of users provided by the server in response to the client's requests. Consequently, an adversary can trivially violate the confidentiality of a conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$ and its underlying group $\mathcal{G}_{\mathsf{proteus}}^{gid}$ by inserting an adversary-controlled client in the list of a user's clients.

Attack Description and Impact

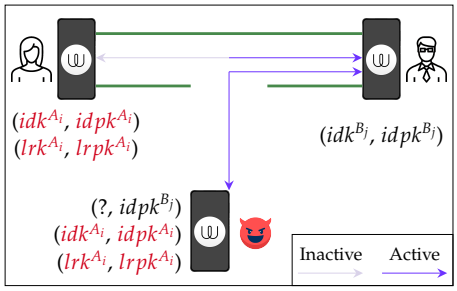
Setting For simplicity, let us assume that users A and B are the only members of conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$, and that both A and B are aware of the same



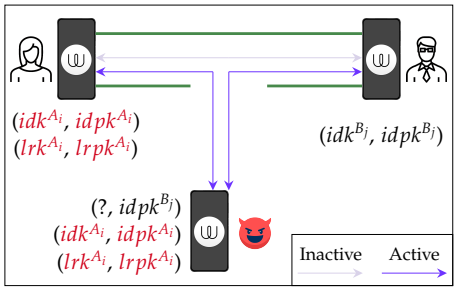
(a) Initial Proteus dialogue between clients of Alice and Bob, before corruption.



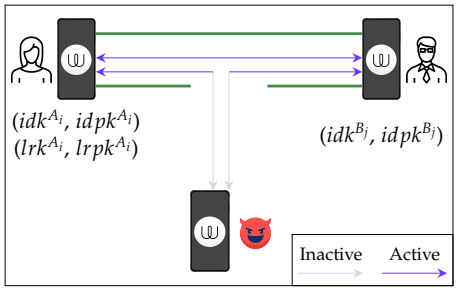
(b) After limited corruption of Alice client's long-term identity and short-term last resort key pairs.



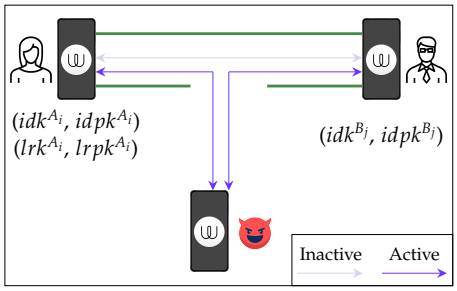
(c) After Disguised Mallory 1 attack malicious session initialization.



(d) After Disguised Mallory 2 attack malicious session initialization.



(e) After reactivation trick. Also assume Mallory loses access to corrupted private keys.



(f) After usage of existing inactive malicious Proteus sessions.

Figure 4.4: Proteus dialogue's state between clients of Alice (A) and Bob (B) during the Mallory in the Middle attack (without loss of generality, we present the attack when only Alice's client is corrupted). With green color we represent the Proteus dialogue. The Proteus sessions it manages are colored purple. The direction of Proteus sessions arrows is interpreted as follows: the client's Proteus dialogue that owns the Proteus session is the opposite to where the arrow head ends up.

membership $\mathfrak{W}^{gid}_{\text{proteus}}$. Moreover, every client A_i of A is aware of a list $\mathcal{C}^{A_i}_{B}$ of B's "honest" clients. Similarly, every client B_j of B is aware of a list $\mathcal{C}^{B_j}_{A}$ of A's "honest" clients. We refer to clients that are not adversarially-controlled as "honest".

Description and Impact Without loss of generality, we describe the attack targeting user A. The adversary M, who controls the server, can inform the clients of A about the addition of a new adversary-controlled client $\widetilde{B}$, either during encryption to a group, as seen in Section 3.6.4, or when clients of A request the list of clients of B, as described in Section 3.3.2. Hence, the list of clients at a client A_i of A will be updated to $\mathcal{C}_B^{A_i} \leftarrow \mathcal{C}_B^{A_i} \cup \{\widetilde{B}\}$. We highlight that A_i unquestioningly trusts the information the server sends without performing any verification through a cryptographic mechanism. Consequently, A_i will establish a Proteus dialogue with the new client $\widetilde{B}$ before sending an encrypted message to group $\mathcal{G}_{\mathsf{proteus}}^{gid}$: the underlying group of conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$. The result of this behavior of A's "honest" clients enables adversary M to read all messages sent from user A to user B in conversation $\mathcal{W}_{\mathsf{proteus}}^{gid}$ (more precisely, in all conversations both A and B are members), thus violating confidentiality by half. In order to enable bidirectional confidentiality violation, M can do the same to "honest" clients of B, but with a new adversary-controlled client $\widetilde{A}$.

Furthermore, we note that M does not inform the "honest" clients of A about the addition of the adversary-controlled client $\widetilde{A}$; otherwise, user A would notice the adversarial activity via a notification pop up in the application of A's clients who are informed about the new addition. The same applies to clients of B for $\widetilde{B}$. Additionally, we see that two different clients are allowed to have an inconsistent view of the group's client membership $\mathfrak{G}_{\mathsf{proteus}}^{gid}$, which is only possible to solve with significant changes to how Proteus groups are handled (addressed directly by MLS).

Thus far, we have observed that the attack is not detectable from the point of view of the "honest" clients of the user whose messages confidentiality is attacked. We now study the behavior of the "honest" clients (of the other user) who are informed about the adversary-controlled client. In Section 3.6.4, we delved into group verification status. If group $\mathcal{G}_{\mathsf{proteus}}^{gid}$ is not verified at a client, then the addition of the new adversary-controlled client does not trigger any notification or warning message that the user can see in the $\mathcal{W}_{\mathsf{proteus}}^{gid}$'s application interface. On the other hand, when group $\mathcal{G}_{\mathsf{proteus}}^{gid}$ is verified, we distinguish two different behaviors.

- If the legacy Android client application is in use, a warning pops up in the conversation and informs the user about the new addition. Then, two options are given to the user: (1) to verify the new client's fingerprint or (2) to acknowledge the risk.
- On the contrary, at the time of writing, if the new Android client application is in use, the addition of the new adversary-controlled client does not trigger any notification or warning message in the conversation. This has catastrophic consequences because the user will not

notice the malicious intervention even if the underlying group was verified.

Mitigations

Our main goal is to prevent adversary M from adding a malicious client to a user's list of clients by going unnoticed, especially in a verified group.

The obvious solution is to at least implement the same behavior in the new Android application as in the legacy Android application. Therefore, the addition of the adversary-controlled client will come to the user's attention.

Alternatively, we recommend a major change in the client registration procedure, which follows WhatsApp's footsteps for device linking. The details are available in WhatsApp's security whitepaper [56]. We now summarize the most important aspects of the solution.

Every user must have a primary client with a long-term identity key pair $(idk^{prim}, idpk^{prim})$, which can link other secondary clients to the user's account. The primary client is responsible for uploading a signed list of clients (client identifiers tied to long-term identity public keys) linked to the user account. The signatures must be generated using idk^{prim}. In addition, the signed client list should contain a timestamp representing the time at which the list was created. The primary device sends the timestamp in the associated data of its encrypted messages to inform other clients (of other users and secondary clients of the same user) about the most recent list they should expect from the server. This is a measure against inconsistent lists of clients and ensures that the server cannot lie about the most recent list.

Moreover, during the client registration procedure, the secondary client generates a long-term identity key pair $(idk^{sec}, idpk^{sec})$ and a secret value s_{link}. The primary client is involved in registration and scans a QR code displayed on the secondary client's screen containing $idpk^{sec}$, and s_{link}. The secret s_{link} is used to integrity protect non-confidential information, e.g., the assigned client identifier, exchanged through the untrusted server (details are omitted). Additionally, the primary client generates the client identifier cid_{sec} for the secondary client. In the current Wire's registration procedure, long-term identity keys are not bound to client identifiers. As a result, pre-key bundles uploaded to the server may have different long-term identity public keys.

In conclusion, the above solution offer many benefits. First, it prevents an adversarial server from responding with arbitrary lists of clients. The signed lists of clients enables only the primary client to generate them. Furthermore, the complexity of group verification is now reduced to simply verifying the primary client's long-term public identity key, which is the root of trust. Undoubtedly, this is a better approach compared to the fragile group verification that Wire currently employes (Section 3.6.4). Last, clients

are no longer independent, which sacrifices usability since client registration is more complicated. Still, it is beneficial since a close, tight bond exists between each user and its clients offering the abovementioned benefits.

4.4.2 Attack 6 (Degraded Forward Secrecy)

In this attack, we look closely at the optional signatures in the pre-key bundles that clients send to the server S_{DS}. In particular, the lack of these signatures, and the absence of mandatory verification by clients and server degrade FS guarantees. The lack of signatures enables an adversary to be actively involved with the choice of the ephemeral/short-term pre-key pair during Proteus session initialization at the initiator. Hence, the key exchange in Proteus can only guarantee weak-Perfect Forward Secrecy (*weak-PFS*) as defined in [39], Section 6.2.

Attack Description and Impact

An important omission in the pre-key bundles sent to the server is the absence of signatures of the public pre-key. Furthermore, even if these signatures were to be provided, the server and client would not verify them regardless. As outlined in Section 3.6.2, the pre-key bundle is crucial for the XPDH key exchange, which establishes a master secret. This master secret, the public pre-key from the bundle, and a fresh ephemeral ratchet key pair chosen by the initiator are used by the initiator to derive message keys for encryption during its first sending symmetric ratchet, starting from chain key $ck^I_{1,1}$.

The attack undermining Forward Secrecy unfolds as follows: Consider an initiator client I of one user and a responder client R of another user. Due to the lack of signatures on the public pre-key in R's pre-key bundles, an adversary M can forge pre-key bundles for R. These forged bundles include R's long-term public key $idpk^R$ and a public pre-key chosen by M. The attacker only lacks R's long-term private key idk^R for these bundles. When I requests R's pre-key bundle, M forwards one of the forged bundles instead of a legitimate one. I considers the forged bundle valid, without any verification, and proceeds with the XPDH key exchange to establish a new Proteus session via in a Proteus dialogue. Consequently, I can encrypt messages to R with this Proteus session using the Proteus dialogue that manages it. However, as the pre-key bundle is forged, M cannot forward encrypted messages to R; doing so would fail initialization from `PreKeyMessage` at R and the malicious intervention is noticed. Instead, M stores the encrypted messages for potential decryption in the future. If M compromises R's long-term key later, it can decrypt these stored messages. Consequently, when Proteus is used in Wire, only a weaker form of Forward Secrecy, that is, weak-PFS is guaranteed.

Mitigations

We propose the obvious mitigation, which mandates other clients (and the distribution server) to verify the pre-key signatures in pre-key bundles. Thus, these signatures must be present in a pre-key bundle and not optional.

4.4.3 Attack 7 (Unauthenticated Metadata)

Figure 3.8 illustrates the contents of an encrypted message using a Proteus session. Notably, certain metadata, including the group identifier and the message timestamp, are not included in the authenticated associated data. However, this information is required by the recipient to correctly display the message within the appropriate conversation and in chronological order, as elaborated in Section 3.6.4. A malicious server can exploit this by tampering with such information, causing confusion for the recipient client regarding the message's order and group/conversation assignment.

Attack Description and Impact

Setting Let us consider a scenario involving two clients: a sender and a recipient. The sender encrypts a message intended for the recipient using a Proteus dialogue, and the sender specifies the group identifier gid for the group $\mathcal{G}^{gid}_{\mathsf{proteus}}$ to which the message is destined. We denote the encrypted message along with its unauthenticated metadata as c_1, while the underlying plaintext message will be denoted by m_1. Later, the sender sends a second message to the same recipient within the same group $\mathcal{G}^{gid}_{\mathsf{proteus}}$. This second encrypted message, along with its unauthenticated metadata, is denoted by c_2, and its corresponding plaintext message is m_2.

Description In the intended secure behavior, when the recipient receives c_1 and c_2 in any order and successfully decrypts them, the outcome is that messages m_1 and m_2 should be displayed in the conversation $\mathcal{W}^{gid}_{\mathsf{proteus}}$'s application interface in the order they were sent within the underlying group $\mathcal{G}^{gid}_{\mathsf{proteus}}$. Furthermore, in a secure E2E communication system, the untrusted server should not have the capability to alter the message order or manipulate the conversation in which they are displayed. Unfortunately, this secure behavior is not guaranteed in Wire.

An adversarial server M possesses the capability to modify both the timestamps and group identifiers of messages since they are not included in the authenticated associated data. Specifically, M can change gid to gid' in both c_1 and c_2, where $\mathcal{G}^{gid'}_{\mathsf{proteus}}$ is a group both clients participate and $gid' \neq gid$. Consequently, the recipient will wrongly display these messages within con-

versation $\mathcal{W}^{gid'}_{\mathsf{proteus}}$ instead of their intended conversation $\mathcal{W}^{gid}_{\mathsf{proteus}}$. Recall that the same Proteus dialogue is shared across all groups in which both clients participate, enabling group "re-assignment".

Furthermore, M can also tamper with the timestamps. If M modifies timestamp t_1 of c_1 such that $t_2 < t_1$, then m_1 will be displayed after m_2, since Wire applications are ordering messages in conversations based on timestamps. This manipulation compromises the correct ordering of messages.

Mitigations

To address the vulnerabilities related to unauthenticated information, we propose two mitigations for each of the concerns. The mitigations for timestamp manipulation are:

1. *Timestamp as Authenticated Data*: We suggest that the timestamp of a message should be included as part of the authenticated associated data, which is protected by the MAC tag of the encrypted message. This approach ensures that only the sender can determine the time at which the message was sent. However, it is important to note that this solution still allows an adversary who later compromises a client to send messages with modified timestamps in the past. This can lead to incorrect message ordering and modification of the message transcript for messages exchanged before the client was compromised.
2. *Double Ratchet Ordering*: If the mitigation strategy against a strong adversary, as described in Section 4.3.2, is implemented, where the dialogue between clients relies on a single Proteus session, then the Double Ratchet algorithm inherently enforces total message ordering in the dialogue. Messages are ordered based on specific rules: (1) messages with a chain key $ck^R_{x,y}$ happen before messages with $ck^I_{x',y'}$ where $x \leq x'$, (2) messages with $ck^R_{x,y}$ happen before messages with $ck^R_{x,y'}$, where $y < y'$, and (3) messages with $ck^R_{x,y}$ happen before messages with $ck^R_{x',y'}$ where $x < x'$. The last two rules also apply for the initiator I's symmetric ratchets. This mitigation can be implemented as an additional measure in groups to, at least, provide correct ordering of messages to the clients that are in a dialogue.

The mitigations for group identifier manipulation are:

1. *Group Identifier as Authenticated Data*: We propose that the group identifier to which a message belongs must be part of the authenticated associated data, protected by the MAC tag of the encrypted message. This approach ensures that only the sender can assign the message to a group. By protecting the group identifier, we prevent unauthorized changes to the destination group.

2. *Distinct Dialogues for Each Group*: Another suggested mitigation is to maintain separate dialogues for each group in which both clients are participating. However, PCS concerning all groups is recovered when all dialogues perform the healing step, in contrast to the current implementation, which has a single Proteus dialogue for all groups.

These proposed mitigations aim to enhance the security and integrity of message timestamps and group identifiers, addressing the vulnerabilities associated with their manipulation.

4.4.4 Attack 8 (Verified Impersonation)

In Section 3.6.3, we described the feature of Proteus dialogue resets, which involve removing the existing Proteus dialogue and initializing a new one. It is essential to note that the attack described below is specific to the Web application. In this attack, an adversary can potentially impersonate one client to another, with their Proteus dialogue used for E2E communication shown as verified.

Attack Description and Impact

Setting Consider a scenario where client A_i of Alice, maintains a verified dialogue $\mathcal{D}_{B_j}^{A_i}$ with client B_j of Bob. The verification process involves Alice confirming out-of-band that Bob's client, B_j, matches the fingerprint of the long-term public key observed during the initialization of dialogue $\mathcal{D}_{B_j}^{A_i}$ at A_i.

Description Now, suppose Alice decides to manually reset the dialogue $\mathcal{D}_{B_j}^{A_i}$ at A_i with B_j. This reset action triggers the initialization of a new dialogue, which requires a pre-key bundle sent by the server. The adversarial server M can exploit this situation by responding to the pre-key bundle request with a malicious bundle. This rogue bundle includes a long-term identity key controlled by M, effectively allowing M to impersonate Bob (B_j) to Alice (A_i). Due to an implementation bug in the Web application, the client does not check whether the same long-term public key was used and, in addition, the verification status of the new Proteus dialogue remains unchanged from the previous one. Consequently, M can impersonate Bob (B_j) to Alice (A_i) in a seemingly legitimate manner, as if the dialogue had been verified. This bypasses the requirement for out-of-band verification of the counterparties identity, which would prevent trivial impersonation by a malicious server.

Mitigations

When a user manually resets a Proteus dialogue, we mandate that the application checks whether the long-term public identity key has changed. If a change is observed, an informative message indicating the change appears in all conversations of the underlying groups the two clients in the Proteus dialogue are members, and the Proteus dialogue status becomes unverified; otherwise, the Proteus dialogue status can be preserved.

4.4.5 Further Findings Affecting Session Independence and PCS

This section highlights two implementation errors that undermine and Post-Compromise Security.

Session Independence

Let us start with the issue related to Session Independence. In both the legacy and new Android applications, the last resort bundle is uploaded only once during the initial client registration and remains unchanged thereafter. As a result, the same short-term last resort key pair is involved in all XPDH key exchanges where the last resort bundle was used to initialize the Proteus sessions throughout a client's lifetime. Essentially, the last resort key functions as another long-term key pair.

This implementation mistake has significant implications. The corruption of both the long-term identity and last resort key pairs allows for decryption of all `PreKeyMessages` of past Proteus sessions initialized from the last resort bundle.

Post-Compromise Security

In the context of Double Ratchet protocols, including Proteus Double Ratchet, recovering security in an E2E communication channel typically occurs when a client, whose full state has been compromised, asymmetrically ratchets forward to its next sending symmetric ratchet. This step involves a new ephemeral ratchet key pair at the compromised client, effectively locking out the adversary. In general, recovering the security of the communication channel means that the adversary can no longer read or insert new messages in the E2E channel.

However, a mistake in the current implementation of Proteus sessions allows an adversary to insert new messages using old sending symmetric ratchets. As previously discussed in Section 3.6.2, a counter indicating the number of messages sent on the previous sending symmetric ratchet is included in the associated data of each encrypted message. Surprisingly, this information is

not utilized during decryption on the receiving end to verify whether a message was encrypted using an older receiving symmetric ratchet, even after transitioning to the next receiving symmetric ratchet (the older symmetric ratchet was used to encrypt a message after it was marked as "terminated"). Consequently, the previous symmetric ratchets can be arbitrarily extended.

This oversight implies that an adversary who gains full access to a client's state can exploit this flaw. By revealing the complete state of a client, the adversary gains access to the chain key for the latest sending symmetric ratchet and can extend it indefinitely. The healing event only occurs once the compromised client has asymmetrically ratcheted forward through five sending symmetric ratchets. At this point, the corrupted receiving symmetric ratchet is finally removed from the last five stored receiving ratchets at the counterparty's session, making the adversary incapable of inserting further messages.

PCS Improvement We propose utilizing the information in the aforementioned counter to effectively "terminate" previous symmetric ratchets. The counter informs the receiver that the previous receiving symmetric ratchet was used to send a specific number of messages; hence, do not decrypt messages that require more symmetric steps of that previous receiving symmetric ratchets than required to derive the message keys of the number of messages specified by the counter. For instance, if $ctr_{prev} = k$ in the associated data of messages encrypted with the $(x+1)^{th}$ sending ratchet of a user U_i in a Proteus session, then the receiver must not derive message keys $(enck^{U_i}_{x,y}, enck^{U_i}_{x,y})$ with $y > k$. We highlight that the receiver can only perform this check after receiving a message on the next receiving symmetric ratchet.

4.4.6 Discussion (Unraveling the Clone Attack Results)

In the study presented in [28], an experiment was conducted to assess the PCS guarantees of various messaging applications, including Wire. This experiment, known as the Clone Attack, aimed to determine whether a clone device could decrypt messages sent after compromise, thus potentially degrading PCS. The comparison of the expected and the actual (in Wire) experimental setups are shown in Figure 4.5.

The Clone Attack proceeds as follows: Two users, Alice and Bob, communicate exclusively through one device each, which we denote by A and B, respectively (as an abuse of our notation for clients and users). The experiment consists of three phases: pre-test, test, and post-test. During the pre-test phase, A and B exchange the same sequence of two `Text` messages five times: a `Text` message from A to B followed by a `Text` message from B to A. After the pre-test phase, a clone of B, denoted by $\widetilde{B}$, is created and disconnected from the internet. The test phase replicates the pre-test

phase, with the same sequence of `Text` message exchanges occurring five times. At the end of the test phase, the original device B is powered off. $\widetilde{B}$ is then reconnected to the internet, initiating the post-test phase. The expected experimental setup is illustrated in Figure 4.5a.

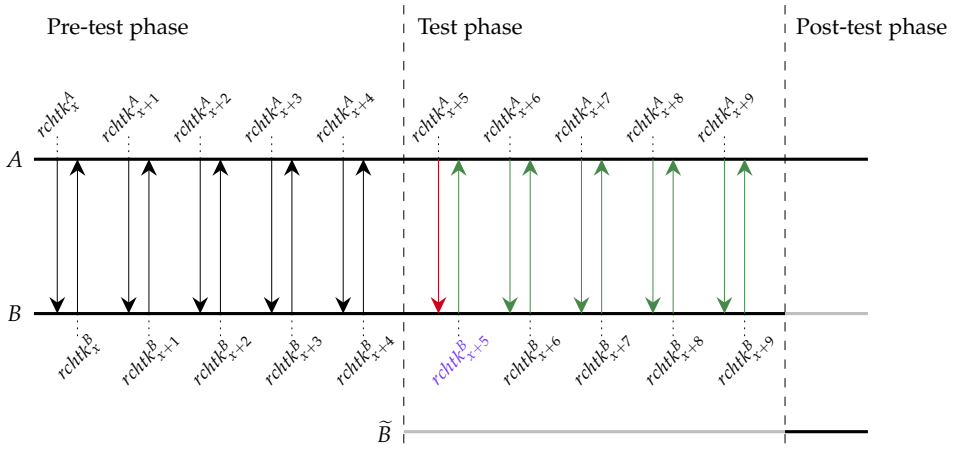
The expectation in an application offering PCS is that $\widetilde{B}$ should be unable to decrypt messages exchanged between A and B during the test phase unless $\widetilde{B}$ has access to the necessary keying material at the time B was cloned. That is, $\widetilde{B}$ should only be able to decrypt messages exchanged between A and B that occurred strictly before the so-called healing step (after cloning). From the analysis of the almost identical Signal Double Ratchet protocol in [25], we know that a requirement for the message keys of a symmetric ratchet to be secure is that either the root key or both parties' ratchet private keys (used for the asymmetric step that results to the symmetric ratchet) are unknown to the adversary. In the Clone Attack, $\widetilde{B}$ is aware of the state of B at the end of the pre-test phase, which contains at least the root key for the next asymmetric ratchet step and the latest ratchet private key of B. Hence, in the Clone Attack, the healing step happens when B selects its next ratchet key pair. Therefore, since B is the last party who sends a message during the pre-test phase (according to the experiment setup), we expect the healing event to occur in two asymmetric ratchet steps from the time B was cloned. As a consequence, $\widetilde{B}$ should be able to decrypt, at most, the messages sent from A in B's next receiving symmetric ratchet, which corresponds to the first `Text` message A sends to B in the test phase of the Clone Attack.

Analyzing the behavior of Wire during the Clone Attack, it was observed that $\widetilde{B}$ successfully received and decrypted the first two `Text` messages sent by A to B during the test phase, thus, including a message after the moment when we would expect the healing step to happen. This behavior seemed to challenge the PCS guarantees of the Proteus Double Ratchet-based sessions used by Wire.

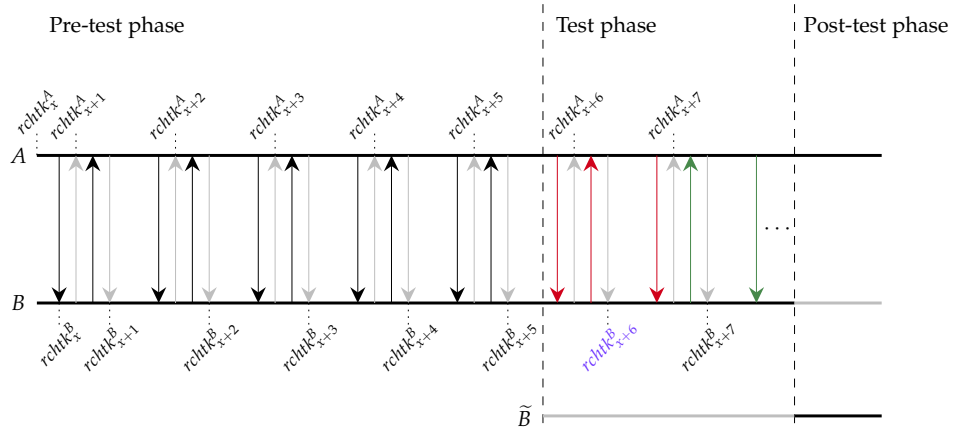
We set out to investigate the underlying causes behind this unusual behaviour observed in the Clone Attack on Wire. Upon studying the Proteus protocol, as discussed in Section 3.6, it was rapidly concluded that $\widetilde{B}$ already possessed the ratchet key pair of B's next sending symmetric ratchet. In the Proteus protocol, when an encrypted message is received, and a Proteus session is utilized for decryption through a Proteus dialogue, the session precomputes the subsequent ratchet key pair when the decryption involves an asymmetric ratchet step forward due to the received message belonging to a new receiving symmetric ratchet. With this knowledge, it becomes clear that if the sequence of message exchanges during the pre-test phase was altered, such that A was the device sending the last message during the pre-test phase and also the first message of the test phase, then B would have precomputed its next ratchet key pair at the end of the pre-test phase. This

would enable $\widetilde{B}$ to decrypt A's first two `Text` messages of the test phase. However, this explanation still didn't align with the expected behavior of the experiment, which assumed that B would be the last party to send a message during the pre-test phase. Additionally, it's worth noting that the experiment disabled typing indications and read receipts (known as read-`Confirmation` in Wire) to prevent hidden messages from affecting the results.

The missing piece of the puzzle was the existence of a third type of hidden message sent within Wire groups known as delivery-`Confirmation`, as described in Section 3.5. While read-`Confirmation` messages were disabled, delivery-`Confirmation` messages remained enabled, without an option for turning them off. Hence, the pre-test and test phases for Wire were effectively altered and satisfied the requirements for enabling $\widetilde{B}$ to decrypt the first two `Text` messages of A during the test phase. Instead of the previous sequence of two messages, a new sequence of four messages was repeated five times: A `Text` message from A to B was followed by a delivery-`Confirmation` message and another `Text` message from B to A, and finally, a delivery-`Confirmation` message from A to B. The actual (in Wire) experimental setup is illustrated in Figure 4.5b.



(a) Expected experiment setup.



(b) Actual (Wire) experiment setup.

Figure 4.5: Experiment setup comparison between the expected and the actual (in Wire). The **black** arrows are `Text` messages, while the grey arrows are delivery-`Confirmation` messages. Moreover, the red arrows shows which `Text` messages of the test-phase the clone $\widetilde{B}$ can decrypt (if they are sent to $\widetilde{B}$ in the post-test phase), while the green are the `Text` messages that are secure after PCS recovery. The purple -colored ratchet key indicates the "healing" step, which recovers PCS from the next sending symmetric ratchet of B and onwards. The horizontal **black** line shows that the client is online, while the grey that it is offline. Last, the dotted lines show which message triggers the generation of each ratchet key pair.

Chapter 5

Analysis of Wire - MLS

In this chapter, we discuss the security implications related to Wire's decisions on MLS usage studied in Section 3.7.3.

5.1 Orchestrator Delivery Service

The orchestrator server S_{DS} is responsible for delivering `Handshake` messages to the group members and resolving conflicting `Commit` messages. Additionally, it has access to all `Handshake` messages content since they are sent in `PublicMessages`. The implication of this choice is that the delivery service S_{DS} can decide which `Commit` is applied by the group members, even to the extent that S_{DS} prevents any of the `Update` proposals and `Commit` messages of a specific member from being implemented. This is because S_{DS} can see the type of a `Proposal`, the content of a `Commit`, and the sender information when clients send `Handshake` messages in `PublicMessage`. Consequently, the S_{DS} can deliberately delay all available mechanisms (`Update` and `Commit`) offering PCS to a particular member of the group. In Wire's MLS implementation, S_{DS} can reject an `Update` proposal and a `Commit` by returning an error indicating to the client that no authorization is granted for this action. The client will then abort the attempt to update (or commit) its information stored in the leaves of the ratchet tree. This shortcoming that defeats PCS when a malicious server suppresses `Update` and `Commit` messages is well known and documented in the MLS standard.

Suggestions A potential mitigation could be to use `PrivateMessages` for the `Handshake` messages, which are confidential between the group members. Recall that `Proposal` and `Commit` messages will use the handshake ratchet, while `Application` messages will use the application ratchet for encryption and decryption. Hence, a `PrivateMessage` still leaks in the associated data the type of the encrypted content, that is, whether it is a `Proposal`,

a `Commit` or an `Application` message. This enables the receiver to decide which ratchet to use for decryption. However, we emphasize that the exact type of the `Proposal` is not leaked through the `PrivateMessage`. Consequently, the server cannot decide whether the content of a `PrivateMessage` is an `Add`, a `Remove`, or an `Update` proposal. Also, the sender's information is encrypted in `PrivateMessage`. Therefore, a malicious server that does not want to block all clients' `Proposal` messages cannot defeat PCS unless a side channel leaks the sender and the exact type.

Moreover, if this mitigation is applied, the server can still resolve conflicting `Commit` messages since the group identifier, the epoch of the group, and the information that the MLS message is a `Commit` are part of the associated data in the `PrivateMessage`, which are not encrypted. Additionally, the server loses access to the content of `Proposal` and `Commit` messages; therefore, authorization checks must be outsourced to the clients. Last, in order to help the server with message delivery, the senders of a `PrivateMessage` can specify the recipients in the associated data because the server will no longer know the group members.

5.2 Out-of-Order Application Messages

In Wire's MLS implementation, clients are configured to maintain secrets from the current epoch as well as the past three epochs. This design decision has implications for the handling of `Application` messages. Specifically, if an `Application` message arrives out of order and falls within one of the past three epochs while the group has transitioned to a new epoch, the application will automatically decrypt and accept it without additional careful examination.

However, this scenario poses a challenge to PCS. In this discussion, we are targeting a PCS form that guarantees that in a group $\mathcal{G}_{\mathsf{mls}}^{gid}$, where the state of the group is maintained for more than one epoch and a client $U_i \in \mathfrak{G}_{\mathsf{mls}}^{gid,e}$, but $U_i \notin \mathfrak{G}_{\mathsf{mls}}^{gid,e+1}$ (the client is removed starting from epoch $e+1$), then messages of client U_i received after the group advanced to epoch $e+1$ must not be accepted even if they are valid messages from epoch e.

In more detail, if an `Application` message encrypted using one of the past three epochs is received from a client who was a group member during that epoch but is no longer part of the group, the application will decrypt and accept the message. Consequently, a removed client, even after being officially removed from the group, can continue to send `Application` messages using the past epoch, since nothing can distinguish delayed messages from messages encrypted after the epoch advancement. These messages will be decrypted and accepted by the application (and therefore, displayed in a

conversation of the underlying group) until the epoch is no longer within the past three epochs. This observation highlights a critical security concern, as the removal of a member does not take immediate effect.

Suggestions We propose that for `Application` messages of one of the three previous epochs, the application has the responsibility of filtering messages sent from old members of the group. This can be achieved efficiently by comparing the client's identity specified in the credential presented at the leaf node of the ratchet tree in the old epoch (indicated in the `Application` message) with the corresponding leaf node in the current epoch. If the identities match, the message is accepted; otherwise, it is rejected.

Additionally, if we want to ensure that all clients in the group will accept the exact same messages of a removed member in the last three epochs, the committer of the epoch, who removes a client with a `Commit`, can include in the associated data of the `PublicMessage` (or `PrivateMessage`, if suggestion in Section 5.1 is implemented) that contains the `Commit` the number of `Application` messages expected on the removed member's application ratchet (for the last three epochs). This must be done for all members removed. Therefore, the committer resolves the ambiguity about whether an `Application` message of a removed member was late or encrypted after the epoch has advanced.

5.3 Credentials

5.3.1 Untrusted Entity

The proposed certificate infrastructure is not resilient enough with Wire's architecture for producing client certificates. The ultimate trust is placed in Wire (controlling both the ACME server S_{ACME} and the distribution server S_{DS}), which should rather be treated as an untrusted entity. Thus, Wire can issue certificates by itself for any client participating in the messaging system and, additionally, for new malicious clients for existing users. These certificates can be used to add arbitrary clients to MLS groups for users with at least one client already member of the group. In Section 3.7, we omitted the details of external commits; here, we specify that clients are allowed to add themselves to the group using the so-called external commits. In Wire's MLS implementation, if a client's user already has at least one client in a group then the client is allowed to perform the external commit.

Moreover, we emphasize that this solution is even worse than the current out-of-band verification because the verification of the certificates will be done automatically using the root of trust: Wire itself. Hence, the MLS groups will be shown verified while, at the same time, Wire can inject an arbitrary number of adversarial clients. However, in contrast to Proteus

groups, the advantage of MLS groups is that all members have a consistent view of the group membership; thus, Wire could not add clients without being noticed by all members.

Suggestions We suggest to implement the countermeasures we propose in Section 4.4.1: users should have a primary client, whose the (long term) signature key pair is used to sign the public (long term) signature key of secondary clients. In the MLS context, the long-term identity key pair will serve as the signature key pair whose public key is present in `KeyPackages` of the client. However, if the signature key pair of the primary client changes, it is required to "re-link" secondary clients and for other users to re-verify the fingerprint of the new public signature key. Another solution could be the additional MLS credentials described in [16], which allow clients to present credentials that associate attributes from OpenID Connect or multiple sources to a signature key pair.

5.3.2 ACME Key Selection

In ACME, an account is used for issuing certificates. Each account is associated with an account key pair whose private key signs each request to the ACME server in order to authenticate the account holders. To issue a certificate, an account holder generates a certificate key pair and sends a Certificate Signing Request (CSR), which requests a certificate binding the public key of the certificate key pair to an identity.

According to the ACME RFC [15] Section 11.1, the account public key and the certificate public key in a CSR must not be the same; otherwise, the repeated use of the same key pair can provide signing oracles to an adversary, which can result in compromise of the ACME account. Wire fails to comply with this requirement as the account key pair is also used as the certificate key pair associated with a CSR[1]. However, despite the inconsistency with the ACME standard, we deem this not to be fully exploitable since both the ACME requests, where the account key pair is used, and the MLS messages, where the certificate key pair is used, have well-defined structures incompatible with each other and therefore achieve a form of domain separation.

[1] https://github.com/wireapp/rusty-jwt-tools/blob/// 443933cea4a984c410b16f90702621c39c4a0bad/acme/src/finalize.rs#L15C25-L15C25

Chapter 6

Conclusion

We conclude by first discussing in Section 6.1 the security status of Wire's E2E protocols in consideration of our analysis of Proteus and MLS in Wire. We then present in Section 6.2 the main lessons learned from the discovered vulnerabilities. Finally, in Section 6.3, we lead the reader to directions for future work.

6.1 Wire's Security Status

In this work, we have analyzed the Wire messenger, particularly its supported E2E protocols. For each of our findings, we provided mitigations that improve the overall security of Wire.

Proteus Proteus is currently the main E2E protocol used for group messaging in Wire. In Chapter 4, we presented attacks on Proteus dialogues undermining Post-Compromise Security despite the fact that Proteus uses the formally analyzed Double Ratchet algorithm with strong-PCS guarantees as a building block. Additionally, we showed that the communication protocol does not guarantee correct ordering and display of messages in conversations. In fact, a malicious server can reorder or even redirect messages to conversations other than the intended destination. Furthermore, we demonstrated that a malicious server can trivially violate the confidentiality of groups due to a naive multi-device solution that treats clients independently rather than collectively. Moreover, we discussed more attacks and findings degrading FS related to ephemeral and short-term values uploaded to the server for the initial authenticated key exchange.

MLS At the time of writing, MLS support in Wire is in an unstable state, but in active development. MLS is a standardized E2E communication protocol that supports out-of-the-box group messaging. The absence of the

requirement for managing groups at the application level and the fact that MLS has undergone extensive security analysis during its standardization process offers a huge improvement to Wire's messaging system. MLS directly addresses most of the security issues of Wire's Proteus E2E communication, including those related to FS and PCS degradation and message redirection to other conversations. However, it also comes with its pitfalls. In Chapter 5, we studied Wire's choices on open decisions in the MLS standard. For instance, we showed that, by maintaining past group state, removing a member from a group does not take immediate effect. Additionally, we discussed how a malicious orchestrator server can delay PCS guarantees, which is not feasible in Proteus. Finally, we argued that client authentication and multi-device support allow the server to trivially violate confidentiality of groups in a way that is even worse than in Proteus groups.

6.2 Lessons Learned

The main lessons learned in this work are twofold.

- *Improper session management can severely impact PCS guarantees*: When dialogues are realized using a session management solution such as the Proteus dialogues, the security guarantees of the underlying building block may not be inherited. In fact, we presented attacks against weak-PCS on Proteus dialogues while the underlying Double Ratchet-based sessions offer strong-PCS. Our findings align with the results presented in [30] for Singla's dialogues, which are implemented using the Sesame session management protocol, functioning similarly to Proteus dialogues. The lesson is that, formally proven secure building blocks do not inherently transfer their security properties when used and interacting with other components at a higher level. Consequently, security needs to be studied at this higher level before deployment.
- *Unreasonable trust to an untrusted component*: The servers controlled by the entity Wire are given ultimate trust. Most of our attacks exploit the absence of cryptographic means for securing the integrity and authenticity of information relayed through or maintained at the untrusted servers. The general rule is that metadata processed at the receiver must be protected from the untrusted intermediate component using a signature scheme or a MAC scheme.

6.3 Future Work

As far as we know, we are the first to study MLS integration in a messaging application. MLS standard leaves many decisions to the application designers. Although the Wire MLS implementation was not stable at the time of

writing this work, we highlighted several decisions and their security implications. In the future, we expect many messengers to adopt MLS, and analyzing their applications will be an exciting area of research. Moreover, when Wire achieves a stable MLS implementation that clients will be using for MLS groups, it will be interesting to revisit and analyze their decisions.

Bibliography

[1] Protocol buffers documentation. https://protobuf.dev/. (Accessed on 08/27/2023).

[2] Timeline of certificate authority failures - sslmate. https://sslmate.com/resources/certificate_authority_failures. (Accessed on 08/27/2023).

[3] Wire - secure messenger - apps on google play. https://play.google.com/store/apps/details?id=com.wire. (Accessed on 09/23/2023).

[4] M. R. Albrecht, S. Celi, B. Dowling, and D. Jones. Practically-exploitable cryptographic vulnerabilities in matrix. In 2023 2023 IEEE Symposium on Security and Privacy (SP) (SP), pages 1419–1436, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[5] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for telegram. In 2022 IEEE Symposium on Security and Privacy (SP), pages 87–106, 2022.

[6] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. Cryptology ePrint Archive, Paper 2019/1489, 2019. https://eprint.iacr.org/2019/1489.

[7] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. Cryptology ePrint Archive, Paper 2019/1189, 2019. https://eprint.iacr.org/2019/1189.

[8] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. Cryptology ePrint Archive, Paper 2021/1083, 2021. https://eprint.iacr.org/2021/1083.

[9] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. Cryptology ePrint Archive, Paper 2020/752, 2020. https://eprint.iacr.org/2020/752.

[10] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Paper 2021/1456, 2021. https://eprint.iacr.org/2021/1456.

[11] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of mls. Cryptology ePrint Archive, Paper 2020/1327, 2020. https://eprint.iacr.org/2020/1327.

[12] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: Analysis of salsa, chacha, and rumba. Cryptology ePrint Archive, Paper 2007/472, 2007. https://eprint.iacr.org/2007/472.

[13] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[14] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[15] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, March 2019.

[16] Richard Barnes and Suhas Nandakumar. Additional MLS Credentials. Internet-Draft draft-barnes-mls-addl-creds-00, Internet Engineering Task Force, July 2023. Work in Progress.

[17] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, Advances in Cryptology — CRYPTO '96, pages 1–15, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[18] Daniel J. Bernstein. Curve25519: new diffie-hellman speed records. https://cr.yp.to/ecdh/curve25519-20060209.pdf, Feb. 09 2006.

[19] Daniel J. Bernstein. The salsa20 family of stream ciphers. http://cr.yp.to/snuffle/salsafamily-20071225.pdf, Dec. 25 2007.

[20] Daniel J. Bernstein. Chacha, a variant of salsa20. http://cr.yp.to/chacha/chacha-20080128.pdf, Jan. 2008.

[21] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Paper 2011/368, 2011. https://eprint.iacr.org/2011/368.

[22] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

[23] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[24] Colin Boyd, Anish Mathuria, and Douglas Stebila. Protocols for authentication and key establishment. https://doi.org/10.1007/978-3-662-58146-9, 2020.

[25] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Paper 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[26] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. Post-compromise security. Cryptology ePrint Archive, Paper 2016/221, 2016. https://eprint.iacr.org/2016/221.

[27] European Commission. Digital markets act. https://ec.europa.eu/commission/presscorner/detail/en/ip_22_6423, Oct. 31 2022. (Accessed on 09/09/2023).

[28] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: Improving post-compromise security in practice. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, page 1481–1495, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. Cryptology ePrint Archive, Paper 2019/477, 2019. https://eprint.iacr.org/2019/477.

[30] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations. Cryptology ePrint Archive, Paper 2022/1710, 2022. https://eprint.iacr.org/2022/1710.

[31] Said Ahmed El-Safadi. Proteus and mls - how will secure communication change? - wire. https://wire.com/en/blog/proteus-and-mls-how-will-secure-communication-change/, Jul. 27 2023. (Accessed on 09/09/2023).

[32] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of google's quic protocol. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, page 1193–1204, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Wire Swiss GmbH. Wire: About us - who we are, our mission, the team, work with us. https://wire.com/en/about/. (Accessed on 09/09/2023).

[34] Wire Swiss GmbH. Wire security whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf. (Accessed on 04/10/2023).

[35] Giles Hogben. Google online security blog: An important step towards secure and interoperable messaging. https://security.googleblog.com/2023/07/an-important-step-towards-secure-and.html, Jul. 19 2023. (Accessed on 09/09/2023).

[36] iiMedia Research. China: Mau of leading messaging apps 2022 — statista. https://www.statista.com/statistics/1062449/china-leading-messaging-apps-monthly-active-users/, Feb. 2023. (Accessed on 09/09/2023).

[37] Tsukasa Ishiguro. Modified version of "latin dances revisited: New analytic results of salsa20 and chacha". Cryptology ePrint Archive, Paper 2012/065, 2012. https://eprint.iacr.org/2012/065.

[38] Jakob Jakobsen and Claudio Orlandi. On the cca (in)security of mtproto. Cryptology ePrint Archive, Paper 2015/1177, 2015. https://eprint.iacr.org/2015/1177.

[39] Hugo Krawczyk. Hmqv: A high-performance secure diffie-hellman protocol. Cryptology ePrint Archive, Paper 2005/176, 2005. https://eprint.iacr.org/2005/176.

[40] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Paper 2010/264, 2010. https://eprint.iacr.org/2010/264.

[41] Signal Messenger LLC. libsignal. https://github.com/signalapp/libsignal. (Accessed on 04/10/2023).

[42] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, Nov. 20 2016. (Accessed on 09/11/2023).

[43] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. https://signal.org/docs/specifications/x3dh/, Nov. 04 2016. (Accessed on 09/11/2023).

[44] Moxie Marlinspike and Trevor Perrin. The sesame algorithm: Session management for asynchronous message encryption. https://signal.org/docs/specifications/sesame/, April 2017. (Accessed on 08/23/2023).

[45] Meta. Messenger secret conversations: Technical whitepaper. https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf. (Accessed on 04/10/2023).

[46] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[47] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger. https://breakingthe3ma.app/files/Threema-PST22.pdf. (Accessed on 03/17/2023).

[48] Kudelski Security and X-41 D-Sec. Wire application level audit (with kudelski security) — x41 d-sec. https://x41-dsec.de/security/report/2018/03/06/projects-x41-wire-phase2/. (Accessed on 09/07/2023).

[49] Kudelski Security and X-41 D-Sec. Wire cryptography audit (with x41 d-sec) - kudelski security research. https://research.kudelskisecurity.com/2017/02/09/wire-cryptography-audit-with-x41-d-sec/. (Accessed on 09/07/2023).

[50] Zhenqing Shi, Bin Zhang, Dengguo Feng, and Wenling Wu. Improved key recovery attacks on reduced-round salsa20 and chacha.

In Proceedings of the 15th International Conference on Information Security and Cryptology, ICISC'12, page 337–351, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] Signal. Technical information. https://signal.org/docs/. (Accessed on 04/10/2023).

[52] We Are Social, DataReportal, and Meltwater. Most popular messaging apps 2023 — statista. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/, Jan. 2023. (Accessed on 09/09/2023).

[53] Telegram. Mtproto mobile protocol. https://core.telegram.org/mtproto. (Accessed on 09/07/2023).

[54] Telegram. Mtproto mobile protocol v.1.0 (deprecated). https://core.telegram.org/mtproto_v1. (Accessed on 09/07/2023).

[55] Viber. Viber encryption overview. https://www.viber.com/app/uploads/viber-encryption-overview.pdf. (Accessed on 04/10/2023).

[56] WhatsApp. Whatsapp encryption overview: Technical white paper. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. (Accessed on 04/10/2023).