Comm

Technical Whitepaper

We describe the security architecture of the Comm platform and its component services. In the following sections, we limit the discussion of system design choices to only those with security or privacy implications.

1 Introduction

This whitepaper introduces Comm, a novel secure messaging protocol. Comm was built with three core goals in mind:

1. Improving the usability–security tradeoff. We expect encrypted messaging to reach mainstream adoption only when users are no longer forced to choose between privacy and a seamless user experience.
2. Enhancing security by leveraging modern technologies such as blockchains.
3. Providing support for federated communities that scale better than existing secure messaging protocols.

Comm borrows design elements from existing messaging protocols, offering end-to-end encryption to protect message content from eavesdropping, forward secrecy to ensure that past communications remain secure even if a user's state is compromised, and post-compromise security, which enables recovery through normal protocol execution.

In the following paragraphs, we discuss Comm's key features that differentiate it from existing systems.

1.0.0.1 Full device encrypted backups. Comm enables users to securely backup their entire messaging history, including chats and media, ensuring seamless data recovery in case of device loss, damage, or upgrade. Full device backup is essential for maintaining continuity, allowing users to preserve their conversations without compromising security.

1.0.0.2 Support for federated communities. Existing end-to-end encrypted messaging protocols do not scale well to support large communities. This is largely due to architectural limitations in how group messaging, key management, and membership state are handled at scale.

Comm addresses this by supporting two distinct messaging protocols. In addition to the end-to-end encrypted DMs protocol used for messaging individuals or groups, Comm also

supports a separate community messaging protocol. This protocol allows users to self-host their own federated communities. Individual hosts have access to plaintext data, and as such clients are able to utilize a more scalable "thin client" model. This approach is somewhat comparable to the classic Internet Relay Chat protocol.

By distributing servers and control among multiple independent hosts, Comm reduces reliance on a single central authority, enhances resilience, and empowers users with greater control over their data. Federated communities promote openness, scalability, and privacy, making the platform adaptable to diverse user needs and organizational policies.

1.0.0.3 End-to-end encrypted notifications. Comm ensures that notifications are end-to-end encrypted. Every notification is end-to-end encrypted before reaching Comm, ensuring that only minimal metadata — such as the sender and recipient device — is exposed.

A key distinction between Comm and other protocols is that Comm includes all necessary notification information in the initial payload. This eliminates the need for a secondary request to fetch additional data, resulting in improved performance, reduced server interaction (and thus fewer opportunities for user tracking), and support for notifications even in environments where the service's servers may be blocked.

Another key difference between Comm and other protocols is that Comm separates content-related keys from notification-related keys. This design stems from Comm's use of iOS Keychain services (5.0.0.1) to store user secrets, which enables a more restrictive access control policy for content keys compared to notification keys.

1.0.0.4 Fully open source. Unlike many secure messaging protocols that are either partially closed-source or rely on opaque backend services, Comm is fully open source — including its clients, servers, and cryptographic libraries. This transparency allows independent researchers, developers, and users to audit the entire system, increasing trust in both the protocol's security guarantees and its implementation [1].

1.0.0.5 Identity validation does not rely on QR code scanning. Many secure messaging protocols rely on peer-to-peer QR code scanning for identity validation, assuming that users will manually scan each other's codes to confirm identities. However, in practice, users often skip this step, leaving identity validation incomplete or insecure. Comm eliminates the need for QR code scanning to authenticate keys, instead leveraging blockchain technology to enhance user verification. By integrating with the Ethereum blockchain, security- and privacy-conscious Comm users can opt to use an Ethereum identity, which peers can independently verify, without needing to manually scan a QR code in-person. This approach not only streamlines the key authentication process but also enhances security by providing a decentralized, tamper-resistant source of trust.

1.0.0.6 Supports group messaging based on pairwise channels. Comm implements group messaging using fully pairwise channels, eschewing a group ratchet protocol. As such, Comm is able to provide strong security benefits by establishing individual encrypted sessions between every pair of participants. This approach offers robust forward

secrecy and post-compromise security, reducing the risk that a single device compromise can incur, at the expense of worse efficiency.

1.0.0.7 Supports web access with an encrypted database. Unlike many end-to-end encrypted messaging protocols that avoid web support due to security concerns, Comm enables secure web access without compromising user privacy. Web access is essential for usability, especially in environments where installing native apps is impractical or impossible. To address the typical concerns around storing sensitive data in the browser, Comm uses encryption-at-rest (5.3) for the local database, and leverages modern capabilities of the Web Crypto API to manage the database encryption key within a secure environment inside the browser context. This approach helps mitigate risks such as key leakage or memory inspection attacks, making secure web access practical and safe.

1.0.0.8 Authentication without phone numbers. Comm relies on password- or wallet-based authentication, rather than phone numbers, significantly enhancing user security and privacy. Phone numbers are inherently insecure as identifiers — they are publicly exposed, easy to guess, and vulnerable to attacks such as SIM swapping, where attackers hijack a user's mobile number to gain unauthorized access to their accounts. In contrast, password- or wallet-based authentication systems remove the dependency on telecom infrastructure and reduce the attack surface.

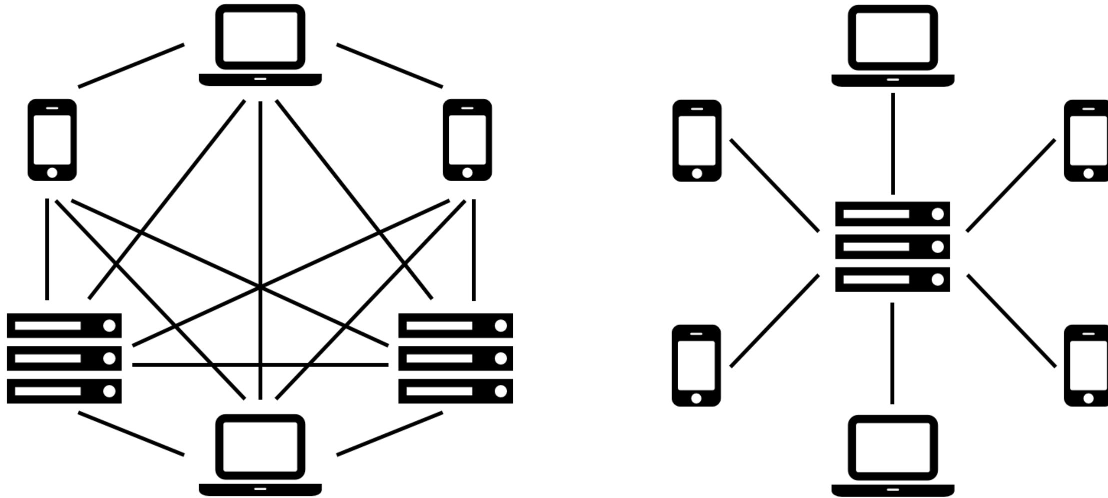
2 Core Concepts

We describe types of user devices and chats supported by Comm.

2.0.0.1 Chats. Comm supports two kinds of chats: direct chats and communities. Communities offer messaging channels (similar to Slack and Discord); direct chats do not. Both kinds of chats support Slack-like threading.

- Direct chats are similar to chats in iMessage, WhatsApp, or Signal and are encrypted using a Double Ratchet [2] session. No single device is considered the host of a direct chat, and instead each user device participates on the same level. See Section 11.1 for details.
- Community related messages, which we refer to as "keyserver-hosted messages" (11.2), are encrypted in transit between the community's hosting keyserver and all user clients using keys derived from Diffie-Hellman agreements. This not only ensures integrity and confidentiality but also allows streamlined authentication (see Section 6.1). Communities are hosted on a keyserver (see below), and rely on that keyserver's query response and background processing capabilities to provide richer functionality than is possible with direct chats. For example, communities support shared applications (e.g., calendars) and consist of discrete messaging channels, similar to those in Discord and Slack.

2.0.0.2 Group Direct Chats. In case of a direct chat involving more than two users, a Double Ratchet session is constructed for every pair of devices used by each participant (Figure 1a).



(a) Network communication pattern in a direct chat with six user devices.

(b) Network communication pattern in a community with six user devices.

2.0.0.3 User Devices. The Comm system consists of two kinds of user devices: clients and keyservers.

2.0.0.4 Clients. Examples of clients include mobile (e.g., iOS, Android), desktop, and web clients. Unlike keyservers, clients are limited in their capacity for background processing, and cannot reliably respond to queries from other devices.

2.0.0.5 Keyservers. A keyserver is a user device that can respond to queries on-demand and execute background processing jobs. A user's keyserver functions as the backend for their applications. Comm does not host user keyservers, but supports various different models for deploying them (e.g., self-hosting, cloud hosting). A user can have up to one keyserver. Communities are always hosted on a keyserver but Comm also supports users running keyservers without any communities on them. A personal keyserver can benefit a user by syncing with all of the user's communities' keyservers in the background. However, not all Comm users need to host a keyserver.

For example, if a user is a member of 10 distinct communities, they need to connect to 10 keyservers when they open the Comm client on their mobile device. If they run their own keyserver, they can connect to just that one, since their keyserver can sync with the other keyservers in the background.

2.0.0.6 Primary Device and Secondary Devices. The initial client device used by a user to register for Comm is termed their primary device. The primary device must be a mobile device. Secondary devices include the keyservers as well as new additional devices that the user can register using the primary device. The primary device can only be switched following the **Restore** (10.2) protocol, and the primary device is the sole device responsible for performing **Backup** (10.1).

3 Comm Services

3.1 Identity Service

Comm's Identity service handles storing and exposing keys for users and devices, in order to allow other users and devices to connect to them. As such, the Identity service handles registration and login for all devices, and is responsible for vending Comm services access tokens, as well as issuing and validating all nonces. It is responsible for storing each user's device list, and other materials necessary for peers to validate a user's chain of trust. In Section 5 we define the keys used by user devices and we explain how trust is boot-strapped using those keys.

The Identity service also tracks each device's version, which is exposed to other devices so that they know what features are supported by the target device. This version is also used in order to determine whether Comm services still support a given device, and the Identity service can inform a given device that an upgrade is necessary.

3.2 Blob Service

Comm's Blob service handles storing arbitrary blobs, as well as indexing them based on an arbitrary key provided by the caller. This service is used for both Message Attachments (11.4) as well as Large Notification Payloads (11.3.0.3). Additionally, the Backup service internally relies on the Blob service for storing large blobs. In most cases, blobs are encrypted, with a couple of specific exceptions:

1. The Blob service is used for community invite links. It maps from a given "slug" to the keyserver that hosts that community, as well as an identifier for the given community.
2. The Blob service is used to identify publicly accessible communities. It maps from a given Farcaster channel identifier to the keyserver that hosts that community, as well as an identifier for the given community.

3.3 Backup Service

Comm's Backup service handles supporting full account restore for a given user. It backs up the primary device's identity keys, as well as all user content not hosted on a specific keyserver. This service is described in more detail in Section 10.1.

3.4 Tunnelbroker Service

Comm's Tunnelbroker service handles three things:

1. It acts as a message broker for Direct Chats (11.1), holding on to encrypted messages for a given device until that device connects and downloads them. Those messages are deleted once the device downloads them and confirms the download.

2. It handles sending notifications to devices, in order to obscure individual device notification tokens from peers, as described in (11.3).

3. Comm supports an unencrypted chat protocol called Farcaster DCs. Tunnelbroker handles connecting and authenticating to the Farcaster service, as described in their docs ([3]). Tunnelbroker stores a given API token for a specific user, and handles forwarding and proxying messages between the Farcaster service and a user's Comm devices.

To avoid an indefinite growth in the amount of queued messages, the Tunnelbroker service may clear messages that haven't been dequeued within a reasonable time.

3.5 Reports Service

The Reports service handles error reports uploaded by devices. These reports are "opt-in": the Comm client defaults to not sending any reports, unless the user goes into their settings and enables this feature. This reporting can be used for debugging issues that Comm users are facing. Before a report is uploaded by a client, sensitive data (such as user messages and identifiers) are redacted from the report.

3.6 Feature Flags Service

The Feature Flags service allows Comm to launch features on clients remotely. Clients talk to this service to determine whether a particular feature is enabled and should be shown to the user.

3.7 Electron Update Service

The Electron Update service allows Comm to publish updates to its macOS and Windows apps, which use Electron ([4]) under the hood.

4 Threat Model

We consider the clients, keyservers, Comm servers, and notification servers – as well as the network channels between them – as potential attack vectors for the adversary.

4.0.0.1 Network.

We can consider an adversary that has full control over a network channel, with the ability to passively observe traffic, actively manipulate it, and inject arbitrary messages. In Comm, all communication channels operate over TLS, while communication between clients is additionally protected by the Double Ratchet layer [2], which is encrypted within TLS.

By observing traffic, the adversary can learn information such as endpoint IP addresses and port numbers, communication timing, message sizes, session initiation patterns, and the frequency of connections. Assuming the security of TLS, the content itself remains private due to TLS encryption. For client-to-client communication, the confidentiality guarantees of Double Ratchet ensure that message content remains private even if the TLS layer is absent or compromised. In such a case, communication would leak only what is exposed by the Double Ratchet protocol itself, including message timing, message sizes, message counts and ordering, and session initiation events.

By manipulating or injecting messages, an adversary may attempt to disrupt the TLS handshake, trigger connection resets, or carry out downgrade or replay attacks. However, properly implemented TLS typically prevents the injection of valid messages into an established encrypted session.

4.0.0.2 Clients.

We can consider a scenario where an adversary temporarily compromises the local state of clients, including their cryptographic keys. This models realistic scenarios such as malware infections, which are particularly relevant given that messaging sessions can be long-lived – sometimes lasting for years – increasing the likelihood of device compromise over the lifetime of a session.

The Double Ratchet algorithm is specifically designed to mitigate the impact of such compromises by providing both forward secrecy and post-compromise security [2]:

1. Forward secrecy ensures that even if an adversary gains access to the current cryptographic state, they cannot decrypt previously exchanged messages, as prior keys are not stored and cannot be reconstructed.

2. Post-compromise security, also known as self-healing, guarantees that once the adversary loses access to the compromised device, future communications will become secure again – typically after the next key exchange between the parties.

The ratcheting mechanism in the Double Ratchet protocol limits the damage from a state compromise to a narrow window of messages and allows the system to recover automatically without requiring user intervention or out-of-band rekeying. Note that, if an adversary compromises the client's local state, they can directly access all messages stored on the device at the time of compromise. Still, thanks to the forward secrecy and post-compromise security provided by the Double Ratchet protocol, both past and future messages – those not present on the device – remain protected.

TLS provides an additional layer of security against client compromise on top of the Double Ratchet protocol. Specifically, TLS ensures forward secrecy across sessions, but not within a single session, and it does not offer post-compromise security either within or across sessions.

4.0.0.3 Keyservers.

In a scenario where keyservers are compromised by an adversary, the attacker could potentially access keyserver-hosted messages. This is because the confidentiality of those messages relies on a TLS layer established between devices and the keyserver (11.2). While TLS provides forward secrecy across sessions, achieving post-compromise security would require keyservers to update their long-term secrets.

For authenticity, keyserver-hosted messages sent by a user device are signed using the device's long-term identity secret key. As a result, any tampered or injected messages will be rejected by the receiving user device (11.2).

Notifications for keyserver–hosted messages are protected by a Double Ratchet session (11.3), providing both forward secrecy and post-compromise security for those messages.

4.0.0.4 Comm Services.

In this section, we'll contemplate a scenario where an adversary is able to compromise some Comm services (3).

User backups are uploaded to Comm only after being encrypted with secret keys derived from the user's password or wallet secret key (10.1). Therefore, their confidentiality and integrity are protected, assuming the security of those keys.

Aside from backup functionality, communication between clients and Comm services predominantly consists of messages that are proxied from other clients. These messages are encrypted using the Double Ratchet algorithm, either directly or indirectly, as in the case of attachments (11.4). The only exceptions are queries for public records, such as identity information.

Next, let's consider how a compromised Comm server can impact the security of Double Ratchet session.

- Ethereum users are protected during session setup against an actively compromised Comm server that is able to manipulate traffic. This is because they can rely on each other's Ethereum public keys (5).
- Password users are only protected during session setup against a passively compromised Comm server (able to see traffic, but not manipulate it). This is because the Comm client has no way to verify the authenticity of the public keys.
- All users are protected *after* session setup due to the Double Ratchet protocol.

By observing the traffic, an adversary can learn all the information typically available to a network-level attacker in the absence of TLS, as well as any metadata leaked by the Double Ratchet layer, as discussed above. However, assuming the authenticity of the protocol initialization phase, i.e., that no injections occurred during that phase, and that the ratchet states have recovered from any prior compromises, the messages exchanged between users remain confidential and authentic – any tampered or injected messages will be detected and discarded by the clients.

4.0.0.5 Notification Servers.

We can consider scenarios where the notification servers are compromised. In such cases, an adversary could gain access to notification messages and potentially manipulate them or even inject new notifications.

By observing notification messages, the adversary learns only basic metadata such as the sending and recipient device, and the timestamp of the notification. Manipulated or injected notifications will be rejected by clients, as Double Ratchet sessions – used for direct chat notifications between clients, as well as between clients and keyservers – protect the authenticity of notifications (11.3).

5 Chain of Trust

In this section we define the keys used by user devices and we explain how trust is bootstrapped using those keys. The list of keys is presented in Table 1.

Key	**Type of Key**
Ethereum (ETH)	`secp256k1`
Content signing (CS)	`ed25519`
Content encryption (CE)	`curve25519`
Content one-time prekeys (COT)	`curve25519`
Content signed prekeys (CSP)	`curve25519`
Notification signing (NS)	`ed25519`
Notification encryption (NE)	`curve25519`
Notification one-time prekeys (NOT)	`curve25519`
Notification signed prekeys (NSP)	`curve25519`
Device list (DL)	JSON array of content signing keys

Table 1: List of keys and their types.

The first entry of Table 1 is the Ethereum key which exists only for users that have associated an Ethereum wallet (5.1). Besides the Ethereum key and the device list key, which are per-user, all the remaining keys are per-device. The chain of trust with respect to the keys listed above is depicted in Figure 2.

The device list for a given user defines the set of devices that user has authorized. The first entry is always the primary device. The primary device signs the device list, thereby authorizing the other devices. We explain the function and benefits of having a primary device in **Reliance on Primary Device** (5.2).

Comm's specification roughly follows the general idea of X3DH [5], that considers long-term identity keys, one-time prekeys, and signed prekeys. However, Comm diverges in two ways, resulting in roughly four times as many keys:

1. We use a second, separate set of keys for notifications, to improve the security of the main set (which we refer to as "content" keys).

2. We use both `ed25519` and `curve25519` keys, rather than one set of `XEd25519` keys. This is due to design decisions made by Vodozemac, the library we use for E2E encryption.

Note that one-time prekeys are optional. If they are not used, the computation follows Signal's specification. However, if one-time prekeys are used, Comm's implementation

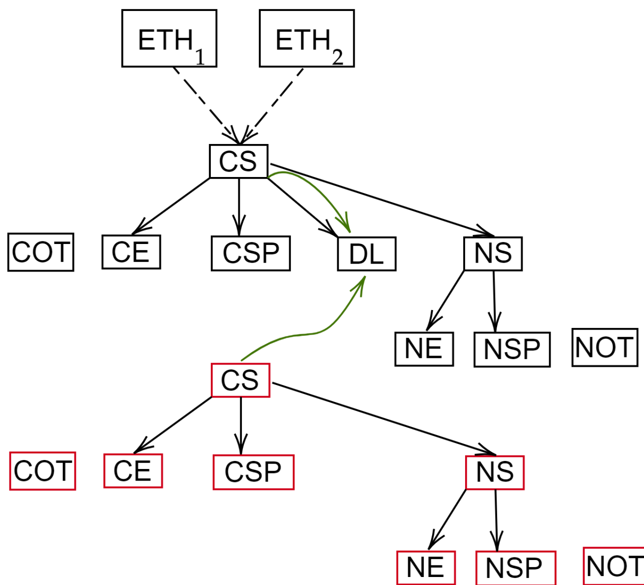


Figure 2: The chain of trust for the keys of Table 2. Keys in black boxes relate to the primary device while those in red relate to a secondary device. A black arrow from one key to another indicates that the former key is signing the latter key. A dotted black arrow indicates that the association only exists for some accounts; namely, users who have associated an Ethereum wallet to their account. A green arrow from a key to the the device list denotes that the key is in the device list. The device list is always signed by the content signing key of the primary device. Finally, keys with no incoming arrows are not signed.

diverges from Signal's specification as follows: it uses the signed prekey in one of the Diffie-Hellman computations and the one-time prekey in two of the Diffie-Hellman computations, while Signal uses the signed prekey twice and the one-time prekey once. This enables backwards compatibility with older versions of Comm, which were using a version of Vodozemac that does not consider signed prekeys and requires one-time prekeys for all handshakes.

The above keys are generated via the **Device Key Generation** protocol, presented in Section 7.1, and uploaded to Comm through the **Key Upload** protocol, described in Section 7.2.

5.0.0.1 Notification keys. The separation between content and notification keys stems from the fact that, for iOS devices, Comm is using iOS's keychain services [6] to store user secrets. Keychain services offer ways to manage the accessibility of individual keychain items according to policies, and this enables Comm to use a more restrictive access control policy for content keys, compared to the one used for notification keys.

In particular, content keys are stored using the "When Unlocked" policy, which enables access of content keys only when the device is unlocked. For notification keys, Comm uses the less restrictive policy of "After First Unlock", which allows access to the keys once the user unlocks the device for the first time after a restart, or if the device does not have a passcode. Access is granted until the device restarts again.

For Android devices, access policies for content and notification keys are the same at this time, but this may change in the future.

Comm implements the separation between content and notification keys using two completely separate Vodozemac accounts.

5.0.0.2 `ed25519` versus `curve25519`. The idea of using both `ed25519` and `curve25519` is explained by Vodozemac in [7]. Whereas Signal's `XEdDSA` spec [8] allows the use of a single long-term identity key for both Diffie Hellman calculations and signing, Vodozemac instead opts for a more traditional approach where `curve25519` keys are used for Diffie Hellman calculations, and `ed25519` keys are used for signatures.

Since the chain of trust is bootstrapped via signatures, we use the `CS` key (`ed25519`) as the primary identity key for a device.

5.0.0.3 Notation. In Table 2 we introduce a second notation that separates public and private keys.

Key	Notation
Ethereum (ETH)	(WalletSecKey, WalletPubKey)
Content signing (CS)	(C.SecKey.Sign, C.PubKey.Sign)
Content encryption (CE)	(C.SecKey.Enc, C.PubKey.Enc)
Content one-time prekeys (COT)	(C.SecKey.OneT, C.PubKey.OneT)
Content signed prekeys (CSP)	(C.SecKey.Pre, C.PubKey.Pre)
Notification signing (NS)	(N.SecKey.Sign, N.PubKey.Sign)
Notification encryption (NE)	(N.SecKey.Enc, N.PubKey.Enc)
Notification one-time prekeys (NOT)	(N.SecKey.OneT, N.PubKey.OneT)
Notification signed prekeys (NSP)	(N.SecKey.Pre, N.PubKey.Pre)
Device list (DL)	RawDeviceList

Table 2: List of keys and their notation.

Note that when we refer to keys focusing on whether they are signing, encryption, one-time prekey, or signed prekeys, i.e., independently of whether they are content or notification keys, we will omit the corresponding prefixes, "C." and "N.", respectively, from the notation.

5.1 Wallet Enrollment

Users may choose to associate an Ethereum wallet with their account. When doing so, the user signs a message containing their Comm public key C.PubKey.Sign using their wallet, thereby generating a cryptographic proof associating their Ethereum wallet to their Comm account. This protocol is described in the Sign In with Ethereum standard defined in Ethereum Improvement Proposal (EIP) 4361 [9].

When using an Ethereum wallet for enrollment, the wallet's signing functionality is used to generate a deterministic signature on a specific message. The Comm mobile apps (iOS and Android) construct the message $\mathsf{msg}_{\mathsf{enr}}$—using a uniformly random nonce provided by the Comm server—and forward the user to the wallet app to procure a signature $\mathsf{sign}_{\mathsf{enr}}$ on it. Replay attacks against Comm servers are repelled by nonce verification.

```
comm.app wants you to sign in with your Ethereum account:
0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2

Primary device IdPubKey: 9nyjNKqi45SVPjiYpOgScoRL4zu36ip7cC5YVt4ZxTU By
continuing, I accept the Comm Terms of Service: https://comm.app/terms

URI: https://comm.app
Version: 1
Chain ID: 1
Nonce: MzI4OTE3NTYzMjg5MTc1Ng==
Issued At: 2022-07-24T16:25:24Z
```

Figure 3: Example message $\mathsf{msg}_{\mathsf{enr}}$ used for enrollment [9].

Figure 3 describes an example message $\mathsf{msg}_{\mathsf{enr}}$. Comm servers store users' wallet address (from which the public key can be derived), enrollment message $\mathsf{msg}_{\mathsf{enr}}$, and the associated signature $\mathsf{sign}_{\mathsf{enr}}$.

5.1.0.1 Protocol: Wallet Enrollment (Start)

- Primary Device requests a nonce from Comm and retrieves its $\mathsf{C.PubKey.Sign}$ (see Section 8)
- Primary Device constructs $\mathsf{msg}_{\mathsf{enr}}$ containing $\mathsf{C.PubKey.Sign}$ and the nonce provided by Comm
- Primary Device asks user to sign $\mathsf{msg}_{\mathsf{enr}}$ using their wallet app to produce $\mathsf{sign}_{\mathsf{enr}}$

5.1.0.2 Protocol: Wallet Enrollment (Finish)

- Primary Device sends the user's wallet address, $\mathsf{msg}_{\mathsf{enr}}$, and $\mathsf{sign}_{\mathsf{enr}}$ to Comm
- Comm confirms the existence of the nonce and that it is at most 120 seconds old
- Comm computes the user wallet's public key $\mathsf{WalletPubKey}$ and verifies that

  $\mathsf{Verify}(\mathsf{WalletPubKey}, \mathsf{msg}_{\mathsf{enr}}, \mathsf{sign}_{\mathsf{enr}}) = \top$

5.1.0.3 Social Proofs of Identity. A valid signature $\mathsf{sign}_{\mathsf{enr}}$ can only be generated using knowledge of the secret key associated with the digital asset wallet. Any Comm user can verify knowledge of the secret key that corresponds to a wallet's public key, $\mathsf{WalletPubKey}$, by verifying that $\mathsf{sign}_{\mathsf{enr}}$ is a valid signature on $\mathsf{msg}_{\mathsf{enr}}$ for $\mathsf{WalletPubKey}$. Therefore, $(\mathsf{msg}_{\mathsf{enr}}, \mathsf{sign}_{\mathsf{enr}}, \mathsf{WalletPubKey})$ forms a public social proof of identity.

Note that only the primary user device generates a social proof of identity. Also note that this signature is distinct from one used for backup in Section 10.1. Finally, note that social proof may have been generated using an outdated primary device content signing key.

5.1.1 Display Names

For users who use an Ethereum wallet as their display name, others will see either the wallet address or a resolved Ethereum Name Service (ENS) name. The resolution from ENS names to Ethereum addresses is handled on Comm clients by third-party providers like Alchemy.

5.2 Reliance on Primary Device

We choose to use a primary device for secondary device login, rather than allowing direct login through password or signature verification, in order to enhance security. This approach offers several advantages, outlined below.

5.2.0.1 Protection against password and signature reuse. If users reuse passwords or wallet signatures across multiple services, an attacker who compromises one service could potentially get access to the user's Comm password or wallet signature. By eliminating the need for passwords or wallet signatures in secondary device login, we significantly reduce the risks associated with such a compromise.

Since secondary devices must be authorized through the primary device, the only action the adversary could perform is executing the **Restore** protocol (10.2) to fully recover the user's primary device. This takeover would be immediately noticeable, as the user would be logged out. In contrast, by allowing password- or wallet-signature-based secondary device login, an attacker could silently add a new device using the stolen credentials, without the user's knowledge.

It's also worth noting that this approach protects against the secret leaking during secondary device authentication to an adversary who is able to see the secret being entered, e.g. by peering over the user's shoulder while they enter a password.

5.2.0.2 Server trust independence. The server does not need to be trusted during the device login process, as the secondary device shares data – including freshly generated keys – with the primary device by scanning a QR code (9.1). This QR code acts as a secure out-of-band channel for key transfer between devices. This reduces the risk posed by potential server compromises.

5.2.0.3 Protection against phishing. In typical password-based registration systems, attackers often attempt to impersonate a trusted service to collect credentials, thereby gaining access to register devices. Comm does not have any websites that prompt users to enter their passwords, and restricts password input to the mobile app only as part of **Restore** (10.2). This approach makes users more likely to recognize phishing sites attempting to request passwords, thereby reducing the risk of successful phishing attacks.

5.3 Encryption At-rest

Comm's local database is encrypted at-rest using SQLCipher [10], ensuring that sensitive user data remains protected even if the device or browser storage is compromised. On Android and iOS clients, the encryption keys are securely stored using hardware-backed secure enclaves using Android Keystore and Apple Keychain services. These platforms provide tamper-resistant environments where cryptographic keys are isolated from the app itself and cannot be extracted, even on rooted or jailbroken devices.

For the web client, Comm uses the Web Crypto API [11] to manage encryption keys within the browser. While not a true hardware enclave, this API allows the generation of non-extractable keys that are isolated from JavaScript and only usable within the browser's cryptographic subsystem. This offers enclave-like protection by ensuring keys cannot be read or exfiltrated by the application, while still enabling secure encryption and decryption of the local database.

6 Authentication

Primitives and Definitions

- TLS is the Transport Layer Security standard, which is used to secure communication between clients and Comm servers
- X3DH refers to the Extended Triple Diffie-Hellman key agreement protocol [5]. Comm's implementation is based on a fork of Vodozemac that adds X3DH [12]. Note that our implementation diverges from Signal's X3DH (Section 5)
- OPAQUE is a password-authenticated key exchange (PAKE) protocol [13]

User devices authenticate either to Comm servers or to keyservers hosting communities. We describe these authentication mechanisms separately. Both authentication protocols ensure forward secrecy and cryptographic deniability. In either case, the corresponding parties compute a shared ephemeral secret that is used to derive an encryption key for the subsequent session.

6.1 Device-to-Keyserver Authentication

The below protocol describes how community keyservers and user devices authenticate each other.

- **User Device** opens an HTTPS (TLS) connection to the keyserver
- The keyserver fetches the client's keys from **Comm**. The key bundle includes the client's long-term identity public key, the signed prekeys together with their signature, and a one-time prekey
- **User Device** fetches the keyserver's keys from **Comm**. The key bundle includes the keyserver's long-term identity public key, the signed prekeys together with their signature, and a one-time prekey

- **User Device** and the keyserver execute X3DH
- The shared key derived from X3DH execution is discarded by **User Device** and the keyserver, and instead a token is assigned to the client by the keyserver, which is used for authenticating the client

More details about how X3DH inputs are stored, retrieved, and validated can be found in Section 5.

In the above protocol the long-term identity keys for Ethereum users are signed by the keys stored in their wallets, and there can be multiple Ethereum wallets signing the long-term identity keys. The process of signing a message using an Ethereum wallet is cumbersome, and Comm is able to avoid prompting the user for additional wallet signatures during day-to-day operations by instead relying on Comm-specific long-term identity keys.

As such, if a user B is using an Ethereum wallet as their display name, then a user A is able to cryptographically verify that the message came from a user who has associated that wallet with their Comm account. On the other hand, if user B is using a Comm username as their display name, no such verification is possible, in which case the long-term identity keys provided by Comm are assumed to be trusted.

This assumption of trust for the long-term identity keys of a non-wallet user is only required during the session initialization phase of the protocol. Unlike traditional E2EE apps like Signal, Comm's approach to backup allows for **Account Restore** 10.2 without requiring a key reset.

The only case where Comm allows key resets is in **Account Reset** 6.3.5, which can occur when a user forgets their backup secret. In this case, all peers of that user receive a prominent notification that alerts them to the change. Such notifications are rare in Comm and are designed to be highly prominent, making key reset attacks more easily detectable by users. This is in contrast to Signal, where such notifications are frequent and occur whenever a peer switches devices, making them more likely to be overlooked by the user.

We eventually want to improve the tamper-proofing of the session initialization phase of the protocol, but as of right now, the Comm client has no way to verify the public keys vended by Comm servers for non-Ethereum users in a tamper-proof way.

Comm does not use mutual TLS (mTLS) for client-keyserver communication in order to avoid implementation complexity. Certificates are also not useful in this setting as immediate revocation is required. Recent OpenSSL vulnerabilities in mTLS highlight the dangers of expanding the attack surface. As X3DH is also used to initiate Double Ratchet sessions for direct chats, this decision enforces uniformity of inter-device communication.

6.2 Device-to-Device Authentication

As already discussed in Section 4, the communication between user devices is protected by the Double Ratchet layer [2]. Assuming the authenticity of the protocol initialization phase – that is, Comm provides the honestly generated users' keys required by the X3DH protocol and no injection attacks occurred during its execution – and further assuming that the ratchet states have fully recovered from any prior compromise, we can conclude that the messages exchanged between users are authentic (i.e. any tampered or injected messages will be detected and discarded by the user devices).

6.3 Authenticating to Comm

In general, Comm services are authenticated by the Comm Services Access tokens (6.3.1). If CSA tokens are not present, but a device has registered its keys with Comm, authentication can be based on known keys (6.3.2).

6.3.1 Authentication Based on Comm Services Access (CSA) Token

Comm issues access tokens to user devices upon successful user authentication (after 7.2), collectively referred to as CSA tokens. CSA tokens are then automatically provided by the client when calling any authenticated Comm RPC. A CSA token is a random base-62 encoded 512-byte value unique to a (user, device) pair. The token is persisted on both the user device and Comm servers, and is used to authenticate to all Comm services. Tokens are never shared, even with the user's other devices (including keyservers), and expire every 90 days. They are refreshed via the **Existing Device Log-in** protocol (9.2).

6.3.2 Authentication Based on Known Keys

1. A device that has already registered its keys with Comm can follow the mechanism outlined in **Existing Device Log-in** (9.2), which uses a challenge-response approach based on known keys.

2. A secondary device that has not yet registered its keys with Comm can authenticate via **Secondary Device Login** (9.1), which relies on the user's primary device scanning a QR code.

6.3.3 Authentication Mechanisms

In three cases, the user must prove their identity from first principles:

1. A user changing their password (6.3.3.7)

2. A user switching from password enrollment to wallet enrollment (6.3.4)

3. A user resetting their account (6.3.5)

Note that for the first two cases password-based authentication is purely an additional layer of authentication for improved security, and never replaces the forms of authentication discussed in 6.3.1 and 6.3.2.

6.3.3.1 Wallet-based authentication

A primary device authenticates to Comm using wallet-based authentication by proving knowledge of the wallet's secret key through the regeneration of the social proof over a uniformly random nonce provided by Comm. This is only used in the extraordinary case of an **Account reset** (6.3.5), initiated by a user contacting a Comm admin. The protocol for this form of authentication is described in the **Wallet Enrollment** (5.1) section.

6.3.3.2 Password-based authentication

Comm uses password authenticated key-exchange (PAKE) for this. Specifically, Comm servers engage user clients in the OPAQUE protocol using an open-source implementation by Facebook [14, 13]. Comm servers do not have access to the plaintext passwords, even during the authentication protocol. Using OPAQUE, users authenticate themselves without ever sending the plaintext password, or any data from which the plaintext password can be derived. Password authentication is also performed before a user changes their password, and before a switch from password enrollment to wallet enrollment (6.3.4).

The protocols below reference structs defined in [14]. At a high level, the protocols generate variables that, either refer to parties' states, or refer to messages sent by the parties. In particular, when **Password Enrollment** starts (6.3.3.3), the client creates `ClientRegistration`, which persists on the client until the final step of client registration completes, as well as `RegistrationRequest`, which is sent to the server. Symmetrically, `ServerRegistration` refers to the state that persists on the server side, while `RegistrationResponse` is the server's response to the client's first message. Finally, `RegistrationUpload`, generated by **Password Enrollment (Finish)** (6.3.3.4) is the client's final message to the server, which the server processes in order to finalize `ServerRegistration`. For more details on the structure of those messages refer to [14].

Note that Comm uses OPAQUE only for authentication and the shared secret derived at the end of the protocol is discarded.

6.3.3.3 Protocol: Password Enrollment (Start)

- Primary Device asks user for the password to construct `RegistrationRequest` and `ClientRegistration`. It then puts aside the `ClientRegistration`, which won't be used again until **Password Enrollment (Finish)** (6.3.3.4)
- Primary Device sends the username and `RegistrationRequest` to Comm
- Comm uses the username and `RegistrationRequest` to produce `RegistrationResponse` and `ServerRegistration`
- Comm sends `RegistrationResponse` to Primary Device

6.3.3.4 Protocol: Password Enrollment (Finish)

- Primary Device uses `RegistrationResponse` and `ClientRegistration` to produce `RegistrationUpload`
- Primary Device sends `RegistrationUpload` to Comm
- Comm uses `RegistrationUpload` to finalize and store `ServerRegistration`

6.3.3.5 Protocol: Password Authentication (Start)

- Primary Device asks user for the password to construct `CredentialRequest` and `ClientLogin`. It then puts aside the `ClientLogin`, which won't be used again until **Password Authentication (Finish)** (6.3.3.6)
- Primary Device sends the username and `CredentialRequest` to Comm
- Comm uses the username and `CredentialRequest` to produce `CredentialResponse` and `ServerLogin`
- Comm sends `CredentialResponse` to Primary Device

6.3.3.6 Protocol: Password Authentication (Finish)

- Primary Device uses `ClientLogin` and `CredentialResponse` to construct `CredentialFinalization`
- Primary Device sends `CredentialFinalization` to Comm
- Comm uses `ServerLogin` and `CredentialFinalization` to authenticate the user

6.3.3.7 Protocol: Change Password

- Primary Device and Comm run **Password Authentication** (6.3.3.5, 6.3.3.6) and **Password Enrollment** (6.3.3.3, 6.3.3.4), in parallel. Each message sent by Primary Device and Comm includes messages from both protocols, therefore, this step requires at most two round trips
- If **Password Authentication** fails, Comm aborts the execution without storing any data sent by Primary Device during **Password Enrollment Registration**
- Primary Device and Comm run **Backup User Keys** (10.1.0.2)
- Comm associates the new `ServerRegistration` with the user

6.3.4 Protocol: Enrollment Switching

- If switching from password enrollment to wallet enrollment,
  1. Primary Device and Comm run **Password Authentication** (6.3.3.5, 6.3.3.6)
  2. Primary Device and Comm run **Wallet Enrollment** (5.1.0.1, 5.1.0.2)
- If switching from wallet enrollment to password enrollment, Primary Device and Comm run **Password Enrollment** (6.3.3.3,6.3.3.4)

6.3.5 Protocol: Account Reset

- The user communicates to a **Comm** admin that they want their account reset using a secure channel of their choosing
- The **Comm** admin performs the following steps:
  - If the account is using password-based authentication (6.3.3.2), then the admin selects a new, secure password, and updates the associated PAKE data
  - For accounts using an Ethereum wallet as their display name, a uniformly random nonce is selected to authenticate the user based on the wallet's secret key (6.3.3.1)
  - The admin invalidates all CSA tokens associated with the account, except for the keyserver if the user has one. This causes all of the user's devices to log out
  - The admin removes all existing backups
  - Finally, the admin communicates the new password or nonce to the user via the aforementioned secure channel
- On the first call to **Restore** (10.2) by **Primary Device**, after **Primary Device** sends the public user identifier, **Comm** responds with `userID` and that no valid BackupID exists
- If the account is using password-based authentication (6.3.3.2), **Primary Device** and **Comm** perform **Password Authentication** (6.3.3.5, 6.3.3.6), where **Primary Device** uses the new password
- If the account is using an Ethereum wallet as its display name, **Primary Device** regenerates the social proof for that wallet using the uniformly random nonce provided earlier, and uploads it to **Comm** (6.3.3.1)
- Similar to **Restore** (10.2), **Primary Device** and **Comm** perform the following steps in a single RPC:
  - Run **Key Upload** (7.2), to upload C.PubKey.Sign that was generated when **Restore** was initiated
  - **Primary Device** sends to **Comm** `userID`, along with a device list signed by C.PubKey.Sign. If the user doesn't have a keyserver, the device list is a singleton [C.PubKey.Sign]. If the user has a keyserver, then the device list also includes the keyserver's content signing key C.PubKey.Sign′: [C.PubKey.Sign, C.PubKey.Sign′]. In both cases, **Comm** saves the device list as the updated SignedDeviceList
  - **Primary Device** and **Comm** perform **Backup User Keys** (10.1.0.2)
  - **Comm** returns a new access token (6.3.1) to **Primary Device**
- **Primary Device** and **Comm** perform **Backup User Data** (10.1.0.3)

- When the restored user adds an old peer and sends them a message, the peer learns that the user's device list has been updated, but the update is not signed by the prior device list's primary device content signing key. An alert is displayed on the peer's device, indicating that the device list is not signed with the primary device's old content signing key

7 Device key and list management

Comm supports multiple devices for a single user. As described in **Chain of Trust** (5), all of a user's devices are represented in a combined device list.

In this section, we describe how keys for new devices are generated, how those keys are uploaded and integrated into the user's device list, and how peer device validate updates to the device list.

We cover these protocols before explaining account registration, as account registration depends on them.

7.1 Protocol: Device Key Generation

Every user device must generate cryptographic key pairs locally. The below steps are executed twice, generating keys for two Vodozemac accounts. The keys of one of the accounts are used as content keys, while the keys of the other account are used as notification keys (see Section 5).

- **User Device** generates long-term key pairs: $(\mathsf{SecKey.Enc}, \mathsf{PubKey.Enc})$, $(\mathsf{SecKey.Sign}, \mathsf{PubKey.Sign})$, short-term signed prekey pairs: $(\mathsf{SecKey.Pre}, \mathsf{PubKey.Pre})$ and sets of 10 one-time prekey pairs: $\{0 \leq i < 10 : (\mathsf{SecKey.OneT}_i, \mathsf{PubKey.OneT}_i)\}$.

- **User Device** generates signed prekey signature $\sigma \leftarrow \mathsf{Sign}(\mathsf{SecKey.Sign}, \mathsf{PubKey.Pre})$

Here we omit the prefixes "C." for content keys and "N." for notification keys, respectively, as we already mention in Section 5. This also holds for the **Key Upload** protocol, presented below.

7.2 Protocol: Device Key Upload

Generated public keys are sent to Comm, along with signed prekey signatures.

- **User Device** sends public keys $\mathsf{PubKey.Enc}, \mathsf{PubKey.Sign}, \mathsf{PubKey.Pre}, \{0 < i < 10 : \mathsf{PubKey.OneT}_i\}$ and signed prekey signature σ to **Comm**

- **Comm** checks that $\mathsf{Verify}(\mathsf{PubKey.Sign}, \mathsf{PubKey.Pre}, \sigma) \stackrel{?}{=} \top$.

- **Comm** stores $\mathsf{PubKey.Enc}, \mathsf{PubKey.Sign}, \mathsf{PubKey.Pre}, \{0 < i < 10 : \mathsf{PubKey.OneT}_i\}$.

```
RawDeviceList = {
    "devices": [
        "6iVZ6YnAZXt8ZmR9IsCHPV1pKss8lu5yUgF6oNxVoHw",
        "4ManHI400UxeQqoTb700qd3se2fg6RAC8CTvrejW4Rs",
        "JpCLbtkVSKGpiMqdlmSD37X2EUYOlfyeT1P5ciL_Ph8"
    ],
    "timestamp": 1699077686000
}

SignedDeviceList = {
    "rawDeviceList": "{\"devices\":[\"6iVZ6YnAZXt8ZmR9IsCHPV1pKss8lu5
        yUgF6oNxVoHw\",\"4ManHI400UxeQqoTb700qd3se2fg6RAC8CTvrejW4Rs\",
        \"JpCLbtkVSKGpiMqdlmSD37X2EUYOlfyeT1P5ciL_Ph8\"],\"timestamp\":
        1699077686}",
    "curPrimarySignature": "UgZP0jLUo5RsOvnfGzcyqgHo_esmNNvnjdL
        b1DOrz8Qbcw5SXL-A_NmOak1jmkfUP2q4aiSOYdYirUKVU-sNDg",
    "lastPrimarySignature": "3T6zpXnuImKaS1lA7TAFGdenR_ApeptQLL-
        e5KanBMWexqtaHJV_TFb3mR6hpN6cqwjsg7b3NWzO5Pi0VeBoAw"
}
```


Figure 4: Examples of RawDeviceList and SignedDeviceList.

7.3 Protocol: Device List Update

A RawDeviceList is a JSON object (Figure 4) consisting of base64 representations of the signing public keys of each device (*devices*) and a millisecond UNIX timestamp. A SignedDeviceList is a JSON object consisting of the stringified RawDeviceList and one or more signatures: *curPrimarySignature* and *lastPrimarySignature*, computed over the RawDeviceList.

The first public key in RawDeviceList[*devices*] is associated with the user's current primary device i.e. the primary device immediately before the **Device List Update**. The *lastPrimarySignature* field is only set in the case of **Primary Device Key Rotation** (8.4) and **Restore** (10.2), as these are the only protocols that change the primary device's C.PubKey.Sign. *lastPrimarySignature* is constructed using a previously used primary device key. Other updates to the device list only consist of *curPrimarySignature*, signed using the user's current primary device.

Note that verification requires knowledge of the previous version of the device list. In particular, the previous primary device's signing public key is needed in order to verify *lastPrimarySignature* in the case of the primary device's C.PubKey.Sign changing. Comm servers verify the timestamp and signature(s), and verify that no more than one device was added and no more than one device was removed. Comm aborts if verification fails.

The primary device then sends a message to all peer devices indicating that the device list has been updated. This includes those devices belonging to the same user, as well as those belonging to that user's peers. Those devices must query Comm to fetch the updated SignedDeviceList, and then update their local copies of it after verification (7.4).

The timestamp allows devices to use the most recent SignedDeviceList and discard previous updates.

- As inputs to this protocol, **Primary Device** is provided the updated RawDeviceList, its new signing key pair ($\mathsf{C.SecKey.Sign}_p$, $\mathsf{C.PubKey.Sign}_p$), and the old signing key pair ($\mathsf{C.SecKey.Sign}_l$, $\mathsf{C.PubKey.Sign}_l$) (if necessary)
- **Primary Device** generates a signature

  $\mathsf{SignedDeviceList}[\mathit{curPrimarySignature}] \leftarrow \mathsf{Sign}(\mathsf{C.SecKey.Sign}_p, \mathsf{RawDeviceList})$
- If necessary, **Primary Device** generates a signature

  $\mathsf{SignedDeviceList}[\mathit{lastPrimarySignature}] \leftarrow \mathsf{Sign}(\mathsf{C.SecKey.Sign}_l, \mathsf{RawDeviceList})$
- **Primary Device** sends an unencrypted message to all peers that an update to SignedDeviceList is pending with timestamp $\mathsf{RawDeviceList}[\mathit{timestamp}]$. Those peer devices then start periodically polling **Comm** for the updated device list
- **Primary Device** sends SignedDeviceList to **Comm**, which retrieves $\mathsf{C.PubKey.Sign}_p$ and verifies:
  - $\mathsf{RawDeviceList}[\mathit{timestamp}]$ is within 300 seconds of current server time and greater than the *timestamp* of the previous update
  - $\mathsf{Verify}(\mathsf{C.PubKey.Sign}_p, \mathsf{RawDeviceList}, \mathsf{SignedDeviceList}[\mathit{curPrimarySignature}]) \stackrel{?}{=} \top$
  - If this update changes the user's primary device, verify that

    $\mathit{lastPrimarySignature} \neq \bot$ and

    $\mathsf{Verify}(\mathsf{C.PubKey.Sign}_l, \mathsf{RawDeviceList}, \mathsf{SignedDeviceList}[\mathit{lastPrimarySignature}]) \stackrel{?}{=} \top$
  - At most one device was added and at most one device was removed via this update
- **Comm** stores SignedDeviceList if verification succeeds and sends a receipt to **Primary Device**
- In their next periodic poll, each peer will then fetch the updated SignedDeviceList from **Comm**

7.4 Validating Device List Updates (from device perspective)

Every user device keeps track of its peers' SignedDeviceLists in order to send and receive messages to those devices. This includes keeping track of the current user's devices.

In order to validate the current state of a particular user's SignedDeviceList, each device will download the history of signed updates between its current version and the latest version. As such, Comm servers must keep a full history of all device list revisions. Recall from 7.3 that during the n-th device list update, the device list is signed both by the content signing key generated during that update and by the content signing key generated during the $(n-1)$-th update. Every user device validates device list history by:

- Verifying that update timestamps are in increasing order
- For each update in the list of updates, let $(\mathsf{C.PubKey.Sign}_l, \mathsf{C.SecKey.Sign}_l)$ and $(\mathsf{C.PubKey.Sign}_c, \mathsf{C.SecKey.Sign}_c)$ denote the content signing key pairs before and after the update, respectively. The device verifies that the update contains valid signatures by executing the following:
  - $\mathsf{Verify}(\mathsf{C.PubKey.Sign}_c, \mathsf{RawDeviceList}, \mathsf{SignedDeviceList}[\mathit{curPrimarySignature}]) \stackrel{?}{=} \top$ where $\mathsf{C.PubKey.Sign}_c$ is the signing public key of the **Primary Device**
  - If this update changes the user's primary device, ensure that $\mathsf{SignedDeviceList}[\mathit{lastPrimarySignature}] \neq \bot$ and $\mathsf{Verify}(\mathsf{C.PubKey.Sign}_l, \mathsf{RawDeviceList}, \mathsf{SignedDeviceList}[\mathit{lastPrimarySignature}]) \stackrel{?}{=} \top$ where $\mathsf{C.PubKey.Sign}_l$ is the signing public key of the **Old Primary Device**

8 Account Management

8.1 Protocol: Account Registration

For password-based authentication (6.3.3.2), the device executes **Password Enrollment** (6.3.3.3, 6.3.3.4), otherwise, executes **Wallet Enrollment** (5.1.0.1, 5.1.0.2). It then runs **Key Generation** (Section 7.1) and constructs a singleton device list consisting of its signing public key. It performs a **Key Upload** (Section 7.2) and **Device List Update** (Section 7.3) in a single RPC.

- **Primary Device** runs **Key Generation** (7.1) to generate $\mathsf{C.PubKey.Sign}$
- **Primary Device** sets $\mathsf{RawDeviceList}[\mathit{timestamp}]$ and $\mathsf{RawDeviceList}[\mathit{devices}] \leftarrow [\mathsf{C.PubKey.Sign}]$
- **Primary Device** and **Comm** perform the following steps in a single RPC:
  - **Primary Device**, for password-based authentication (6.3.3.2), executes **Password Enrollment** (6.3.3.3,6.3.3.4), otherwise, executes **Wallet Enrollment** (5.1.0.1,5.1.0.2)
  - **Primary Device** and **Comm** run **Key Upload** (7.2) and **Device List Update** (7.3)
  - **Comm** generates a unique `userID` using UUID version 4 [15] and returns it to **Primary Device**

8.2 Protocol: Account Deletion

A user may delete their Comm account using their **Primary Device**. In the following protocol, each **Secondary Device** except for the keyserver, resends the account deletion message to peers to protect against a race where a new peer is only known to a particular **Secondary Device**. As such, each peer device will receive an account deletion message from each of the deleted user's devices.

- **Primary Device** sends an account deletion message to each **Secondary Device** associated with the user

- **Primary Device** and each **Secondary Device**, except for the keyserver, send an account deletion message to each peer device

- Each **Secondary Device** logs out

- **Primary Device** calls **Comm** to delete its account and **Comm** invalidates all CSA tokens associated with the account

- **Primary Device** logs out upon receiving confirmation from **Comm**

8.3 Protocol: Primary Device Log-out

A user may log out from the primary device, in which case all of the secondary devices, except for the keyserver, are invalidated.

- **Primary Device** makes sure there is no ongoing backup of UserKeys and UserData. If there are any backup operations pending, including pending log uploads, then **Primary Device** waits for those uploads to complete, as defined in **Backup User Keys** (10.1.0.2) and **Backup User Data** (10.1.0.3)

- **Primary Device** sends a log-out message to each **Secondary Device** associated with the user, except the keyserver

- Each **Secondary Device** except the keyserver visually logs out and clears its database

- **Primary Device** sets $\mathsf{RawDeviceList}[\mathit{timestamp}]$. If the user has a keyserver, **Primary Device** sets $\mathsf{RawDeviceList}[\mathit{devices}] \leftarrow [\mathsf{C.PubKey.Sign}, \mathsf{C.PubKey.Sign}']$, where $\mathsf{C.PubKey.Sign}'$ is the keyserver's content signing key. Otherwise it sets $\mathsf{RawDeviceList}[\mathit{devices}] \leftarrow [\mathsf{C.PubKey.Sign}]$.

- **Primary Device** and **Comm** run **Device List Update** (7.3)

- **Comm** invalidates all CSA tokens associated with the user's devices, except for the keyserver if the user has one, and sends a confirmation to **Primary Device**

- Upon receiving confirmation from **Comm**, **Primary Device** visually logs out and clears its database

8.4 Protocol: Primary Device Key Rotation

Once every few months, a user's primary device keys are rotated. This involves **Key Generation** (7.1), a **Device List Update** (7.3), and **Key Upload** (7.2). In this case, **Device List Update** requires two signatures: *lastPrimarySignature* is generated using the old signing key while *curPrimarySignature* is generated using the new signing key.

After key rotation, all Double Ratchet sessions are reinitiated and the backup is regenerated using new key material. Additionally, if the account is using an Ethereum wallet as its

display name, the corresponding social proof is also regenerated. In the following example, the primary device switches its signing public key from $\mathsf{C.PubKey.Sign}_o$ to $\mathsf{C.PubKey.Sign}_n$.

- **Primary Device** removes $\mathsf{C.PubKey.Sign}_o$ from $\mathsf{RawDeviceList}[\mathit{devices}]$
- **Primary Device** runs **Key Generation** (7.1) to generate $\mathsf{C.PubKey.Sign}_n$
- **Primary Device** sets $\mathsf{RawDeviceList}[\mathit{timestamp}]$ and inserts $\mathsf{C.PubKey.Sign}_n$ at the beginning of $\mathsf{RawDeviceList}[\mathit{devices}]$
- Run the following steps in a single RPC:
  - **Primary Device** and **Comm** run **Key Upload** (7.2)
  - **Primary Device** and **Comm** run **Device List Update** (7.3), which handles setting $\mathsf{SignedDeviceList}[\mathit{curPrimarySignature}]$ and $\mathsf{SignedDeviceList}[\mathit{lastPrimarySignature}]$
  - If the account is using an Ethereum wallet as its display name, **Comm** sends a uniformly random nonce to **Primary Device**. **Primary Device** then regenerates the social proof for that wallet (5.1.0.3) using the uniformly random nonce, and uploads it to **Comm**
- **Primary Device** (using rotated keys) and **Comm** run **Existing Device Log-in** (9.2). **Comm** associates the existing CSA token (6.3.1) with the new keys
- **Primary Device** runs **Double Ratchet Reset** (11.1.2) to reinitiate Double Ratchet sessions
- **Primary Device** sends a message via **Send Message** (11.1.4) to all peers informing them that messages queued for the old primary device should now be sent to it
- **Primary Device** runs **Backup User Keys** (10.1.0.2)

9 Device management

9.1 Protocol: Secondary Device Login

Every user device must link itself to the user account. For secondary device login, there must already be a primary device registered with Comm that corresponds to the user.

The secondary device uses a QR code to share data with the primary device. If the secondary device is a keyserver, the QR code is displayed in its console output. In that QR code, the secondary device includes a 256-bit AES secret key sk, which is used to secure communication between them during the protocol. The QR code also includes the secondary device's signing public key $\mathsf{C.PubKey.Sign}$. This QR code represents a deep link to the Comm mobile app of the form:

```
comm://qr-code/<URI-encoded JSON>
```

This allows users to scan the QR code via any camera application. The passed JSON object contains sk and $\mathsf{C.PubKey.Sign}$ as:

$$\{\mathtt{aes256} : \mathsf{sk}, \mathtt{ed25519} : \mathsf{C.PubKey.Sign}\}$$

The primary device generates a $\mathsf{RawDeviceList}$ that includes the secondary device, and runs a **Device List Update**. The secondary device is then able to authenticate to Comm services and restore user data. $\mathsf{BackupDataKey} = (\mathsf{BackupDataKey.Comp}, \mathsf{BackupDataKey.Log})$ is a pair of symmetric keys used to encrypt user data (Section 10.1).

- **Secondary Device** runs **Key Generation** (7.1) to generate $\mathsf{C.PubKey.Sign}$
- **Secondary Device** generates random 256-bit AES key sk and displays $(\mathsf{sk}, \mathsf{C.PubKey.Sign})$ as a QR code
- **Primary Device** reads $(\mathsf{sk}, \mathsf{C.PubKey.Sign})$ via the QR code
- **Primary Device** sets $\mathsf{RawDeviceList}[\textit{timestamp}]$ and appends $\mathsf{C.PubKey.Sign}$ to $\mathsf{RawDeviceList}[\textit{devices}]$
- **Primary Device** and **Comm** run **Device List Update** (7.3)
- **Primary Device** packages `userID`, $\mathsf{BackupID}$, and $\mathsf{BackupDataKey}$ together, and encrypts them using sk. It then informs **Secondary Device** of the successful **Device List Update** by sending the ciphertext
- **Secondary Device** and **Comm** run **Key Upload** (7.2) and **Existing Device Log-in** (9.2)
- **Secondary Device** sends `userID`, $\mathsf{BackupID}$, and the new CSA token from the prior step to **Comm**, and **Comm** responds with ct_2 (10.1)
- Let ct_2' be the part of ct_2 which corresponds to the encryption of $\mathsf{UserData}$ (10.1.0.3). **Secondary Device** decrypts user-generated data as $\mathsf{UserData} \leftarrow \mathsf{AEAD.Dec}(\mathsf{BackupDataKey.Comp}, \mathsf{ct}_2')$ and inserts it into the local database
- The remaining part of ct_2 is a list of SQLite logs, encrypted under $\mathsf{BackupDataKey.Log}$ (10.1.0.3). **Secondary Device** decrypts logs one by one using $\mathsf{BackupDataKey.Log}$ and applies the changes to the local database

9.2 Protocol: Existing Device Log-in

The protocol is run whenever CSA tokens expire, or as part of **Primary Device Key Rotation** (8.4) and **Secondary Device Login** (9.1). It allows existing user devices in the device list to authenticate to Comm using a challenge-response mechanism, which helps to prevent replay attacks. This protocol does not require user interaction and can be used by all user devices. The challenge message sent by Comm to client is a uniformly random nonce, n. The device signs the nonce and sends the signature to Comm. Comm verifies the signature and issues a valid CSA token to the device if verification succeeds.

- **Existing Device** retrieves ($\mathsf{C.SecKey.Sign}_e$, $\mathsf{C.PubKey.Sign}_e$) and connects to **Comm**

- **Comm** responds with a uniformly random nonce, n

- **Existing Device** computes signature $\sigma_L \leftarrow \mathsf{Sign}(\mathsf{C.SecKey.Sign}_e, n)$ and sends σ_L to **Comm**

- **Comm** checks the existence of the nonce and that it is at most 120 seconds old. If the check fails, abort

- **Comm** checks $\mathsf{Verify}(\mathsf{C.PubKey.Sign}_e, n, \sigma_L) \stackrel{?}{=} \top$, and log-in succeeds if and only if verification succeeds

- **Comm** issues a valid CSA token (6.3.1) to **Existing Device** if log-in succeeds

9.3 Protocol: Secondary Device Log-out

Secondary devices can remove themselves from an account. The secondary device will need to rely on the primary device to update the device list.

- **Secondary Device** sends a log-out message to **Comm**

- **Secondary Device** sends a log-out message to **Primary Device**

- **Primary Device** sets $\mathsf{RawDeviceList}[timestamp]$ and $\mathsf{RawDeviceList}[devices] \leftarrow [\mathsf{C.PubKey.Sign}]$

- **Primary Device** and **Comm** run **Device List Update** (7.3)

- **Comm** invalidates the CSA token associated with **Secondary Device** and sends a confirmation message to **Secondary Device**

- **Secondary Device** visually logs out and clears its database

9.4 Protocol: Secondary Device Removal

In Comm, a secondary device can only be removed by the **Primary Device**. When a device is removed, the user's existing **Primary Device** and **Comm** run **Device List Update** (7.3). In the following protocol, a secondary device with signing key $\mathsf{C.PubKey.Sign}$ is removed.

- **Primary Device** sets $\mathsf{RawDeviceList}[timestamp]$ and removes $\mathsf{C.PubKey.Sign}$ from $\mathsf{RawDeviceList}[devices]$

- **Primary Device** and **Comm** run **Device List Update** (7.3)

- Removed device logs out upon receiving confirmation from **Primary Device** and **Comm** invalidates the associated CSA token (6.3.1)

9.5 Replenishing One-Time Prekeys

One-time prekeys, used during key exchange as part of X3DH [5], are replenished by each client device when there are fewer than 6 corresponding unused key pairs stored by Comm. . The device generates and sends a new batch of 20 one-time prekeys. Comm prevents *drainage* attacks on one-time prekeys by imposing rate limits on users initiating sessions.

9.6 Rotating Signed Prekeys

Signed prekeys are used during key exchange as part of X3DH [5]. Comm clients rotate their signed prekeys periodically for good cryptographic hygiene.

- **Primary Device** generates prekey pairs $(\mathsf{SecKey.Pre}, \mathsf{PubKey.Pre})$ and their signatures, as in **Key Generation** (7.1)
- **Primary Device** uploads the new signed prekey pairs and their signatures to **Comm**, as in **Key Upload** (7.2)

10 Resilience

Primitives and Definitions

- $\mathsf{AEAD.Enc}$ and $\mathsf{AEAD.Dec}$ represent encryption and decryption functions of an authenticated encryption scheme (e.g., AES-GCM).
- $\mathsf{argon2i}(k, m)$ is a key derivation function (KDF) designed for hashing a password m with key k [16]. We use 256-bit digests for use as AEAD keys.
- Keygen, $\mathsf{Sign}(\mathsf{sk}, m)$, $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ represent a secure digital signature algorithm (e.g., EdDSA, ECDSA). Keygen returns a (private key, public key) pair and $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = \top$ if and only if $\sigma = \mathsf{Sign}(\mathsf{sk}, m)$ is a valid signature on message m.
- $\mathsf{BackupKey}$ is a symmetric encryption key used to encrypt $\mathsf{UserKeys}$
- $\mathsf{UserKeys}$ consists of $\mathsf{BackupDataKey}$ and the primary device's content signing (CS) key. For simplicity we actually serialize the primary device's entire content Vodozemac `Account`, which also includes the content encryption key (CE) as well as one-time prekeys (COT) and signed prekeys (CSP). However, we only need to use the content signing (CS) key in **Restore** (10.2), and as such $\mathsf{UserKeys}$ is not updated when prekeys and one-time prekeys are updated.
- $\mathsf{UserData}$ includes user-generated data including messages and media. It does not include ratchet state (11.1.1).

10.0.1 Journaling

Every update sent to a user device is considered a message and appended to a journal. If the update is part of a direct chat thread, it can only be decrypted by user devices participating in the thread. Otherwise if the update is part of a community, the keyserver hosting the community decrypts the update and makes it available to mobile, desktop, and web clients used by members of the community.

10.0.2 Backup encryption

Comm backups are all encrypted on the client, in order to guarantee that Comm does not have access to the user's content.

Backups can be encrypted using either a password or a signature generated by an Ethereum wallet. Note that this signature is private and never leaves the device, and as such must be distinct from the public signature described in Section 5.

10.1 Backup

The backup service allows a user to store an encrypted copy of their data on Comm's servers. UserKeys and UserData and are backed up separately using BackupKey and BackupDataKey respectively. UserData can be restored easily on new devices as it is encrypted separately (Section 9.1). Note that UserKeys contains BackupDataKey. Only the primary user device can perform **Backup**. Backups upload encrypted copies of the users' keys, UserKeys, and data, UserData.

Below we present the protocol **Backup User Keys**, that creates a backup of UserKeys and the protocol **Backup User Data**, that creates a backup of UserData. For users who have secured their backup with an Ethereum wallet, during its first execution, **Backup User Keys** generates the backup message on its first step. We formalize this below as the **Backup Message Generation** protocol.

We separate **Backup User Keys** and **Backup User Data** in order to make Primary Device Key Rotation cheaper. Primary Device Key Rotation requires an update of UserKeys, but we avoid an update of UserData, which is much larger.

10.1.0.1 Protocol: Backup Message Generation

For users who have secured their backup with an Ethereum wallet, execute the following:

- **Primary Device** constructs m_B as depicted in Figure 5. Note that the nonce is randomly selected and not used for validation
- **Primary Device** asks the user to sign m_B using their wallet app to produce a deterministic signature $\mathsf{Sign}(\mathsf{sk}, m_B)$. The signature never leaves **Primary Device**

10.1.0.2 Protocol: Backup User Keys

```
comm.app wants you to sign in with your Ethereum account:
0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2

By continuing, I accept the Comm Terms of Service:
https://comm.app/terms

URI: https://comm.app
Version: 1
Chain ID: 1
Nonce: ODQzOTA1NDA4NDM5MDU0MA==
Issued At: 2022-07-25T16:25:24Z
```

Figure 5: Example message m_B used for backup key generation [9].

- During the first protocol execution, for users who have secured their backup with an Ethereum wallet, Primary Device performs **Backup Message Generation** (10.1.0.1) and retrieves the new m_B
- Primary Device chooses random $\mathsf{BackupID}$ and sets

$$\mathsf{BackupKey} \leftarrow \begin{cases} \mathsf{argon2i}(\mathsf{BackupID}, \mathsf{password}) & \text{if backup secured with password} \\ \mathsf{argon2i}(\mathsf{BackupID}, \mathsf{Sign}(\mathsf{sk}, m_B)) & \text{if backup secured with wallet} \end{cases}$$

- Primary Device sends $\mathsf{BackupID}, m_B, \mathsf{ct}_1 \leftarrow \mathsf{AEAD.Enc}(\mathsf{BackupKey}, \mathsf{UserKeys})$ to Comm
- Comm stores $(\mathsf{BackupID}, m_B, \mathsf{ct}_1)$

10.1.0.3 Protocol: Backup User Data

- Primary Device chooses random $\mathsf{BackupDataKey}$
- Primary Device and Comm perform the following steps in a single RPC:
  - Primary Device sends initial compaction to Comm

$$\mathsf{ct}_2 \leftarrow \mathsf{AEAD.Enc}(\mathsf{BackupDataKey.Comp}, \mathsf{UserData})$$

  - Comm stores ct_2
  - Primary Device and Comm perform **Backup User Keys** (10.1.0.2)
- On an ongoing basis, Primary Device listens for updates to its local database via the SQLite session extension [17]. These updates are termed logs, and are encrypted using $\mathsf{BackupDataKey.Log}$, and then uploaded to Comm, which appends them to ct_2

There is a distinction between the computation of so-called compactions and logs. The former refers to an initial compression of UserData at the time that the **Backup User Data** is first executed, while the latter refers to logs tracking subsequent changes to UserData. Our protocol uses SQLite's session extension capability [17] to track changes over the SQLite database.

The SQLite database is encrypted at-rest (5.3) using BackupDataKey.Comp, and as such the database file itself serves as the compaction. In contrast, logs are individually encrypted with the BackupDataKey.Log.

Note that **Backup User Data** can be executed via calls made by other protocols, such as **Restore** (10.2), as well as on its own whenever a new backup compaction needs to be generated for performance reasons.

10.1.0.4 Protocol: Change Backup Secret

In order to change their backup secret, password users perform **Change Password** (6.3.3.7). On the other hand, users who secure their backup with an Ethereum wallet will perform the following steps:

- Primary Device performs **Backup Message Generation** (10.1.0.1) and retrieves the new m_B
- Primary Device and Comm perform **Backup User Keys** (10.1.0.2) using the new m_B

10.2 Restore

Comm does **not** have the backup keys and **cannot** access these data. However, the user can regenerate BackupKey at a later time, using their password password or secret Ethereum signing key sk. This allows the user to decrypt UserKeys, which restores BackupDataKey. Subsequently, BackupDataKey is used to decrypt UserData, as well as any logs that were appended. Cryptographic keys for the restored primary device are regenerated.

Note that for accounts using an Ethereum wallet as their display name, the social proof must be regenerated, so that peers can verify the chain of trust.

Protocol: Restore

- Primary Device runs **Key Generation** (7.1) to generate new keys including a signing public key C.PubKey.Sign
- Next, the Primary Device needs to determine the restoring user's public user identifier. Primary Device prompts the user to select from two options:
  - For accounts that use an Ethereum wallet as their display name, the wallet address is the public user identifier. To get the wallet address, Comm prompts the user to connect their wallet. Additionally, Comm and Primary Device follow the procedure defined in 5.1 in order to generate a new social proof: Comm provides

a uniformly random nonce to **Primary Device**, which includes it in a message that is signed by the user's wallet

 - For all other accounts, the public user identifier is a **Comm** username, which can be entered via a text field

- **Primary Device** calls **Comm** with the public user identifier to get BackupID, ct_1, `userID`, as well as the device list. If the user has a keyserver, the device list contains its content signing key, $\mathsf{PubKey.Sign}'$. For backups that are secured with an Ethereum wallet, m_B is also included in the response

 - If a user does not have any backups, then they will be unable to perform a restore. This scenario can happen from a race condition where **Primary Device** fails to complete an initial backup after **Account Registration** (8.1), as well as in the case of a **Account Reset** (6.3.5). We treat both cases as an account reset; see 6.3.5 for more details

- For backups that have been secured with an Ethereum wallet, **Comm** prompts the user to connect their wallet, e.g., by scanning a QR code via WalletConnect, and then requests them to compute a signature over m_B, $\mathsf{Sign}(\mathsf{sk}, m_B)$), where sk and m_B, refer to the secret signing key and backup message, respectively, used by the user in their latest call to **Backup User Keys** (10.1.0.2)

- **Primary Device** computes BackupKey as

$$\mathsf{BackupKey} \leftarrow \begin{cases} \mathsf{argon2i}(\mathsf{BackupID}, \mathsf{password}) & \text{if backup secured with password} \\ \mathsf{argon2i}(\mathsf{BackupID}, \mathsf{Sign}(\mathsf{sk}, m_B)) & \text{if backup secured with wallet} \end{cases}$$

 In the above, password refers to the password used by the user in their latest call to **Backup User Keys** (10.1.0.2), if that backup is secured with a password

- **Primary Device** decrypts cryptographic state as $\mathsf{UserKeys} \leftarrow \mathsf{AEAD.Dec}(\mathsf{BackupKey}, \mathsf{ct}_1)$ and retrieves BackupDataKey

- **Primary Device** and **Comm** compute in a single RPC the following:

 - Run **Key Upload** (7.2)
 - **Primary Device** sends to **Comm** `userID`, along with a device list with a current timestamp signed by both $\mathsf{C.PubKey.Sign}$, as well as the old primary device's content signing key (from UserKeys). If the user doesn't have a keyserver, the device list is a singleton $[\mathsf{C.PubKey.Sign}]$. If the user has a keyserver, then the device list also includes the keyserver's content signing key $\mathsf{C.PubKey.Sign}'$: $[\mathsf{C.PubKey.Sign}, \mathsf{C.PubKey.Sign}']$
 - **Comm** performs the signature verification step of the **Device List Update** protocol (7.3), and if verification succeeds, saves the device list as the updated SignedDeviceList and sends a receipt to **Primary Device**
 - **Primary Device** and **Comm** perform **Backup User Keys** (10.1.0.2)

  - Comm stores a new backup that consists of the new BackupID and ct_1 generated during **Backup User Keys** (10.1.0.2), as well as the ciphertext ct_2 from the backup being restored
  - If the account is using an Ethereum wallet as its display name, Primary Device uploads the new social proof for that wallet, generated during the earlier step when determining the user's public user identifier
  - Comm invalidates all existing access tokens associated with the user, except for the keyserver's token, if such a token exists
  - Comm returns a new access token (6.3.1) to Primary Device

- Primary Device sends `userID`, the new BackupID, and the new CSA token to Comm, and Comm responds with ct_2

- Let ct_2' be the part of ct_2 which corresponds to the encryption of UserData (10.1.0.3). Primary Device decrypts user-generated data as $\mathsf{UserData} \leftarrow \mathsf{AEAD.Dec}(\mathsf{BackupDataKey.Comp}, \mathsf{ct}_2')$ and inserts it into the local database

- The remaining part of ct_2 is a list of SQLite logs, encrypted under BackupDataKey.Log (10.1.0.3). Primary Device decrypts logs one by one using BackupDataKey.Log and applies the changes to the local database

- Primary Device sends an unencrypted message to all peers that the SignedDeviceList has been updated. The peers then fetch the updated SignedDeviceList from Comm

- Upon realizing the new device list reflects a change in primary device, the peer devices will resend any messages queued for the old primary device to the new one via **Send Message** (11.1.4)

10.2.0.1 Privacy. Comm learns BackupID. If the user used password authentication, BackupID does not reveal any information about BackupKey to Comm without knowledge of password. Similarly, Comm cannot forge $\mathsf{sign}_{\mathsf{backup}} = \mathsf{Sign}(\mathsf{sk}, \mathsf{msg}_{\mathsf{backup}})$ without knowing sk and hence cannot compute BackupKey if wallet enrollment is used (see Figure 5). All communication is secured by TLS.

11 Chat Messages

11.1 Direct Chats

As noted in Section 4, all user devices running clients can access community data stored on that community's keyserver. In case of direct chats between users A and B, a Double Ratchet session is constructed for every pair of devices used by A and B. Message content is then encrypted once for each recipient device.

11.1.1 Double Ratchet

Comm uses a fork of Vodozemac, an open-source implementation of the Signal Double Ratchet protocol [12, 18, 19]. We describe some subtle differences between Comm, Vodozemac, and Signal here.

- Signal and Comm use the Extended Triple Diffie-Hellman (X3DH) protocol [5] while Vodozemac uses Triple Diffie-Hellman (3DH). The primary difference between Comm's fork of Vodozemac and the upstream version is our decision to implement Signal's X3DH, which was made in order to improve forward secrecy and cryptographic deniability. Note that our implementation diverges from Signal's X3DH (Section 5).

- Signal uses the XEdDSA and VXEdDSA Signature Schemes that enable the use of the same key for both elliptic curve Diffie-Hellman and signatures. Vodozemac and Comm use separate `ed25519` and `curve25519` keys [7]. While we may choose to implement XEdDSA and VXEdDSA in the future, we chose not to prioritize it initially.

11.1.2 Protocol: Double Ratchet Reset

During **Primary Device Key Rotation** (8.4), the primary device needs to reset its Double Ratchet sessions with all of that user's other devices, as well as all of that user's peer users' devices.

- Primary Device deletes ratchet state for other device

- Primary Device initiates an X3DH handshake [5] with the other device

11.1.3 Errors in Decryption

Errors may arise in decryption due to multiple reasons. In Comm, messages that cannot be decrypted correctly are typically resent by the sender. There are two exceptions described below, in which case the messages are ignored by the recipient.

- **Ignored by Recipient**

  There are only two cases where messages that fail decryption are ignored:

  1. Subsequent account deletion messages, received after the first such message, cannot be decrypted due to deletion of ratchet state

  2. Messages or notifications received after the user has logged out cannot be decrypted due to inaccessibility of ratchet state

- **Resent by Sender**

  In all other cases, messages that fail decryption will be resent by the sender. Here are a couple examples:

  1. Messages that cannot be decrypted due to internal Vodozemac errors

2. Messages that cannot be decrypted due to a concurrent **Primary Device Key Rotation** (8.4)

In such cases, the recipient initiates an X3DH handshake in order to perform a **Double Ratchet Reset** (11.1.2). The sender always attempts to resend messages after a successful reset. All messages attempted by the sender after the last message received from the recipient are resent.

Note that **Primary Device Key Rotation** (8.4) itself entails a **Double Ratchet Reset** and another reset is not performed in the second case.

11.1.4 Protocol: Send Message

Assume a user A on client C who wishes to send a message to a group of users in the list $\mathcal{B}$.

- The client device C generates a list of devices $\mathcal{D}$ that need to receive the message. This includes all other devices of the current user A, as well as all devices of all users in the list $\mathcal{B}$
- For each device in $\mathcal{D}$, C retrieves from Comm the corresponding X3DH input keys and initiates an X3DH [5] session with that device if one does not exist yet. X3DH sessions are initiated on-demand whenever users wish to send messages. More details about how X3DH inputs are stored, retrieved, and validated can be found in Section 5
- If during the execution of X3DH session setup, a device gets an error while decrypting a message from another device (11.1.3), this may indicate that both devices are attempting to initiate an X3DH session. In that case, users terminate the session that has been initiated by the user with lower `userID`, in lexicographic order, and messages that fail decryption are resent by the sender
- For each device in $\mathcal{D}$, send the message to that device using the Double Ratchet session

Note that there is a race condition that can occur between **Restore** (10.2) and X3DH session initialization. If an X3DH session is initialized but doesn't make it into the restoring user's backup, then the other party will not receive a device list update by the restoring user when they conclude the restoration protocol. To mitigate this risk, all devices will periodically poll for the current device list of any users from whom they haven't heard from recently. Since this risk can only occur during primary device restoration, and since that protocol logs out all secondary devices, we can assume that any message received from the peer device indicates that this race condition has not occurred.

11.1.5 Out-of-Order Messages

Comm's architecture is built to handle out-of-order messages.

- For direct chats, we rely on the Double Ratchet protocol, which can handle some finite number of out-of-order messages by preserving skipped ratchet keys.

- Comm uses a pub/sub architecture in a couple places: for client-to-client communication, as well as for client communication with the Tunnelbroker service (3.4). In these cases, we avoid checkpointing in favor of individual message confirmations to avoid the risk of losing an out-of-order message.
- In certain cases, an individual client-to-client message may rely on an earlier message. As an example, you can imagine a chat being created, and then a message being sent to that chat. Comm handles these cases by queuing up the second message in cases where it cannot be processed, and only processing it after preconditions are met due to the first message being processed.

11.2 Keyserver-Hosted Messages

Keyserver-hosted messages are transmitted over a TLS connection between devices and keyservers. These messages are generated on user devices, with delivery managed by the keyserver. For authenticity, each message sent by a user device is signed using the device's long-term identity secret key, C.SecKey.Sign. The receiving user device verifies the signature using the corresponding public key, C.PubKey.Sign, and upon successful verification, processes the incoming message. Keyserver-hosted messages are held indefinitely.

11.3 Notifications

Operating system and browser vendors maintain notification servers and channels that enable Comm to alert users when updates are made. Comm supports such notifications for iOS, Android, macOS, and Windows devices, as well as generally for all browsers supporting the Notifications API ([20]).

The notification servers of operating system and browser vendors handle the delivery of notifications, acting as a sort of proxy. In order to identify a particular instance of the Comm client app running on a particular device, the vendor provides a notification token to the client app, which is then shared with Comm.

Comm services (3) persist the device's notification token. When a new user device comes online, it shares the notification token with Comm services, and the latter stores a map from the device's public key to the notification token. This enables Comm to avoid sharing the notification token with each user's client, which would allow a malicious client to arbitrarily send notifications to a target.

The content of the notification is opaque for Comm services, which proxies the notification to the notification servers. The notification is encrypted before it reaches Comm services, and they are only aware of basic metadata such as the sending and recipient device. The sender must encrypt the notification once for each recipient user device. If the notification fails to send, it is not persisted.

11.3.0.1 Encryption ratchets for direct chat notifications. Each pair of communicating parties maintains a Double Ratchet session and notifications are sent to recipients through the associated sessions initiated with their notification keys (5.0.0.1). Upon receiv-

ing an encrypted notification, a user device uses the Double Ratchet session associated with the sender to decrypt the notification content.

11.3.0.2 Encryption ratchets for notifications of keyserver-hosted messages. Notifications for keyserver-hosted messages are encrypted using the Double Ratchet session established between the keyserver and the user device using their notification keys (5.0.0.1). In order to make sure that DH ratchet steps are performed despite the fact that communication is one-way (devices do not send notifications to keyservers), the user device periodically sends dummy messages to the keyserver across the Double Ratchet session.

11.3.0.3 Large notification payloads. If the size of the encrypted notification is larger than that allowed by the notification service provider, the sender encrypts the payload using a 256-bit AEAD key and uploads the encrypted blob to the Comm server. The sender sends a shorter encrypted notification containing the AEAD key and a SHA-256 hash of the encrypted blob.

11.4 Message Attachments

11.4.0.1 Size. To preserve privacy, all small files (smaller than five MB) are padded to the nearest multiple of ten KB using PKCS#7 padding. Larger files are not padded.

11.4.0.2 Encryption. A new AES key sk is generated for each attachment sent over a direct chat. The attachment data are encrypted using sk and the encrypted blob is uploaded to the keyserver. The recipients are sent a SHA256 hash of the blob and sk. Message recipients retrieve the blob using the SHA256 hash and decrypt it using sk.

11.4.0.3 Forwarding. If a user forwards a received attachment, the SHA256 hash of the encrypted blob and the decryption key are sent to the recipient.

References

[1] Comm Team. Comm implementation. *GitHub*. Available: https://github.com/comme2e/comm/.

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

[3] Farcaster. Farcaster docs. *Farcaster Documentation*. Available: https://docs.farcaster.xyz/.

[4] Electron. Electron auto updater. *Electron Documentation*. Available: https://www.electronjs.org/docs/latest/api/auto-updater.

[5] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Signal Inc.* Available: https://signal.org/docs/specifications/x3dh/.

[6] Apple Inc. Restricting keychain item accessibility. *Apple Developer*. Available: https://developer.apple.com/documentation/security/keychain_services/keychain_items/restricting_keychain_item_accessibility#overview.

[7] Matrix Foundation. Signature keys and user identity in libolm. *GitLab*. Available: https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/signing.md.

[8] Trevor Perrin. The xeddsa and vxeddsa signature schemes. *Signal Inc.* Available: https://signal.org/docs/specifications/xeddsa/.

[9] Wayne Chang, Gregory Rocco, Brantly Millegan, and Nick Johnson. Eip-4361: Sign-in with ethereum. *Ethereum Improvement Proposals, no. 4361*. Available: https://eips.ethereum.org/EIPS/eip-4361.

[10] LLC Zetetic. Sqlcipher. *SQLCipher Documentation*. Available: https://www.zetetic.net/sqlcipher/documentation/.

[11] MDN contributors. Web crypto api, cryptokey: extractable property. *Web Crypto API Documentation*. Available: https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey/extractable.

[12] Comm. Vodozemac (fork). *GitHub*. Available: https://github.com/CommE2E/vodozemac.

[13] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: an asymmetric pake protocol secure against pre-computation attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2018.

[14] Facebook. The opaque key exchange protocol. *GitHub*. Available: https://github.com/facebook/opaque-ke.

[15] Leach Davis, Peabody. Universally unique identifier version 4. *Internet Engineering Task Force*. Available: https://datatracker.ietf.org/doc/html/rfc9562#uuidv4.

[16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[17] SQLite. The session extension. *SQLite Documentation*. Available: https://sqlite.org/sessionintro.html.

[18] Matrix Foundation. Vodozemac. *GitLab*. Available: https://github.com/matrix-org/vodozemac.

[19] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *Signal Inc.* Available: https://signal.org/docs/specifications/doubleratchet/.

[20] WHATWG. Notifications api. *WHATWG Standard*. Available: https://notifications.spec.whatwg.org/.