Veracity: End-to-end Encrypted Messaging with ***Forward Secrecy*** and ***Post-Compromise Security***

WANG Yuqi YANG Xikun YANG Jinkun

CUI Mingyue GAO Ningcong

***Abstract*—We present the *Veracity*, an end-to-end encrypted (E2EE) messaging system that far exceeds project requirements. X3DH and Double Ratchet are implemented to provide forward secrecy and post-compromise security. Client-side secrets such as prekeys and ratchet states are encrypted at rest via OS-keyring-backed encryption AES-256-GCM-SIV with per-purpose HKDF derivation. For authentication we pair Argon2id-hashed password with TOTP two-factor authentication. Collectively, they provide security grantees far beyond what password-only and static-key baselines can offer.**

Beyond implementation, formal verification of 11 security properties across auth, session, and device-key lifecycles are performed using TLA+ and Alloy [1], [2]. We further adopted NIST statistical test suite to verify in-transit data randomness. Overall, our project amounts to 20,000 lines of Python code.

I. INTRODUCTION

A. What We Built

This report presents *Veracity*, a desktop E2EE messenger app. All *baseline features* as required by this project have been *completely* fulfilled, which we discuss below, along with any *advanced features* that we implemented. We included a full requirements fulfillment checklist can be found in the Section X and a full analysis of the security advantages offered by the advanced features at Section VI.

Baseline Security Features. Ordinary messaging system relies on a trusted server. However, since *Veracity* operates under an Honest-but-Curious (HbC) model, we implemented E2EE 1:1 messaging with server-side ciphertext store-and-forward. As per requirement by the project, we also implemented timed self-destruct messages, a friend request workflow (also opaque to server) with anti-spam measures, delivery status indicators (delivered / read), conversation list with unread counter badge, hashed password, and OTP authentication. All of the above are sent over TLS-secure transport [3].

Advanced Security Features. Since the specification permits *any* protocol that is appropriate under the HbC threat model, we push this to the extreme, and implemented the full *Signal Cryptographic Stack*[1]. Most notably, we implemented:

- **X3DH**: async Diffie-Hellman (DH) key agreement [4]
- **Double Ratchet** for forward secrecy and post-compromise recovery [5].

All client-side secrets (identity keys, ratchet states, X3DH prekeys, and session tokens) are encrypted at rest via AES-256-GCM-SIV with per-purpose HKDF key derivation [6], [7], all of which backed by **OS Keyring**. Authentication utilizes Argon2id password hashing, paired with **Time-based One-Time Password (TOTP)** two-factor authentication for maximum account security [8], [9].

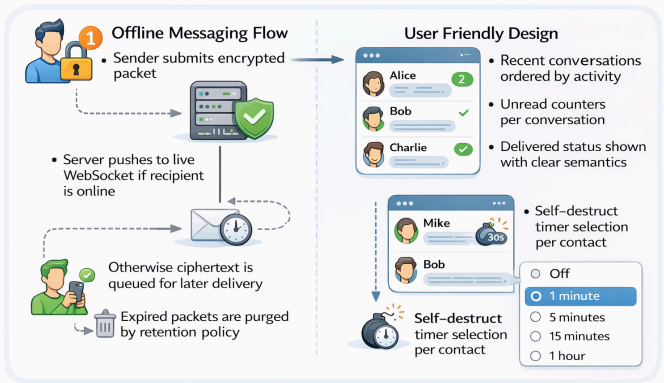

Fig. 1. Illustration of user interaction logic with our *Veracity* messenger. We offer intuitive and user-friendly UI/UX.

User Interface. We built a chat UI with an early MacOS Aqua style. Firstly, event notifications including sending failures, message destruct timer updates, message rejections, and device fingerprint change notifications are all visible in the main interface, instead of hiding them in logs. Secondly, timestamps for newly-arrived messages, system notifications, accept/reject labels, and friend request prompts are all shown directly in the message timeline. Moreover, our UI/UX is intuitive. For example, when a message is rejected, users are provided with next-step guidance (e.g., option to resend a friend request), instead of just an error banner. In terms of program structure, we use a GUI-Controller-Service architecture. The GUI does not directly interact with business logic. Instead, everything is managed by the Controller, which makes the user interface system easier to maintain.

Evaluation. For evaluation we utilized TLA+/Alloy to formally verify client-server security [2], and use MITM interception and NIST 800-22 randomness statistical test suite to verify the client-client message confidentiality.

[1]https://signal.org/docs/

B. Key Security Properties

Veracity provides arguably the highest-possible level of security attainable given the time limit of this project. Specifically, we provide the following security properties, most of which arise from the aforementioned *advanced features*:

1) **Forward Secrecy.** Defends against *Past Key Compromise*: even if long-term private key was stolen, ***past messages are safe***, which is not possible with traditional key transport. This is primarily attained through ephemeral key agreement (DH) and immediate deletion of shared secret after deriving message key; the KDF chain in Double Ratchet also contributes [4], [5].
2) **Post-Compromise Security.** Defends against *Current Key Compromise*: even if security keys are compromised now, ***future messages are safe***, which is not possible with traditional static shared key.This is attained through Double Ratchet's *slow ratchet*: each new DH ratchet step introduces completely resets shared secret. Thus, any attacker who obtained a past key is quickly locked out of future messages within a single turn of message [5].
3) **Replay and Reorder Resistance.** Defends against *Network Attacker Replay/Reorder*: even if attacker replays old ciphertext, or reorders messages in transit, ***duplicates are rejected and ordering is preserved***. Double Ratchet's per-message ratchet counter is inherently resistant to message replays. Reordering is also countered by maintaining a skipped-message window (up to 1000 skipped message per KDF chain). Furthermore, the use of One-time Prekey (OPK) also makes our system resistant to X3DH initialization replay [4], [5].
4) **Message Length Confidentiality.** Defends against *Ciphertext Length Leakage*: even if ciphertext were known, ***original message lengths cannot be reverse-engineered exactly***. This is attained through 0x80-then-zero padding to 80-byte block boundaries, following Signal's production parameters [10].
5) **Nonce-Misuse Resistance.** Defends against *Implementation-Level Nonce Reuse*: even if a nonce is accidentally reused, ***confidentiality degrades gracefully***, which would have been catastrophic with ordinary AES-GCM; this is attained through using AES-GCM-SIV; attacker learns only whether two plaintexts are identical, and is unable to recover plaintext directly. This is attained through AES-256-GCM-SIV, which provides the SIV (Synthetic IV) construction on top of standard AEAD [6].
6) **Encryption-at-Rest.** Defends against *Physical Device Theft*: even if the local database got exfiltrated, ***local secrets remain encrypted*** as decryption requires OS keyring. This is attained through per-purpose HKDF key derivation via OS-keyring-guarded root secret with context-bound associated data [7]. This prevents malicious actors from transplanting ciphertext across terminal.

II. THREAT MODEL

A. Adversary Model

We adopted *much* ***stricter*** threat models than what the three listed in the project specification.

Honest-but-Curious (HbC) Sever. The server executes the protocol faithfully, but attempts to learn as much as possible within these constraints. Specifically, it can observe username, user relations, logout events, active-device state, prekey upload/claiming, message packet metadata, ciphertext size, and traffic timing. It cannot, however, decrypt message payloads of gain access to user's locally encrypted client secrets. The server cannot even differentiate the type of data that is being sent (e.g., regular message or friend request).

Network Attacker. We account for an almighty network attacker controlling the entire communication channel, and can eavesdrop, delay, drop, replay, reorder, duplicate, inject, and any other arbitrary actions given that they are feasible within current technological limits. This would mean that the attacker *cannot* break standard cryptographic primitives or correctly configured TLS channels [3].

Attempt Unauthorized Access. We assume that attacker will attempt password guessing, OTP guessing, credential, and bearer-token theft. *Veracity* address this with Argon2id password hashing with salting to prevent rainbow-table attacks [8], TOTP-based MFA to make OTP guessing more challenging [9], session expiry and revocation to limit damage of token theft.

Partial Client-side Compromise We accounted for two types of client-side adversaries: 1) an adversary presented on the user's device (e.g., malware) but has *limited* system privilege, and 2) an adversary that was able to *temporarily* gain access to the user's current cryptographic states or *permanent* access to previous states. The former is addressed by sealing local secrets behind admin-privileged, keyring-backed encrypted storage; the latter is addressed through the post-compromise security and forward secrecy offered collectively by X3DH and Double Ratchet [4], [5].

B. Trust Assumptions

Trust on First Use. We assume that the server follows the specified protocol and database logic faithfully during the *first interaction*. For example, when establishing with another user, the server is assumed to be trustworthy and provide authentic recipient identity key and prekey bundles [4]. It's worth nothing though, even if the server is malicious on first use, said inconsistency can still be detected through client-side fingerprints [4]; however, this serve more as a last resort than standard procedure, hence we cannot claim resistance.

Secure Cryptographic Primitives. We assume that the security primitives used in this project are all correctly and securely implemented in the Python `Cryptography` library. This includes, namely, X25519, Ed25519, HKDF-SHA256, HMAC-SHA256, SHA-256, Argon2id, TOTP, and AES-256-GCM-SIV [6], [7], [8], [9].

Secure OS Keyring. We also assume that all OS default keyring are secure and trustworthy. Note, we explicitly detect and reject untrusted keyring backends, so our trust assumption is specifically narrowed to OS-default keyring implementations.

Secure Transport Layer. Client should be able to securely authenticate into the intended server endpoint. We assume that TLS is correctly configured and deployed, such that client-server communication is protected against eavesdropping and MitM attacks [3].

Atleast One Uncompromised Client. While we assumed in Section II.A that attacker can compromise client secrets, it is limited to just one party. Our end-to-end confidentially requires that the tw communication endpoints are *not* both compromised. Our forward secrecy and post-compromise security do *not* defend against an attacker who can continuously control both parties.

C. Out-of-scope Items

This section details the security statements that we do ***not*** claim. Following Section II.B, the following out-of-scope items can be direclty inferred:

- Fully malicious or compriomised server.
- Insecure cryptographic primitives or implementations.
- Insecure OS keyring implementations.
- Insecure TLS configuration or deployment.
- Continuous biparty client compromise.

We additional note these out-of-scope items that are not directly implied by our trust assumptions:

- Zero-knowledge privacy policy
- Side-channel attacks
- Denial-of-Service (DoS) attack
- Formal proof of the entire messenging stack.

III. ARCHITECTURE

A. Trust Boundaries

Trust boundaries mark where data moves between components operating under different trust assumptions. We define the following components in our system architecture:

- **Trusted Client**: hold plaintext & cryptographic states.
- **HbC Server**: relays ciphertext data (w/ metadata).
- **OS Keyring**: protects client secrets at rest.
- **Untrusted Network**: hostile transport environment.
- **Server Storage**: queue for ciphertext and metadata.
- **Client Storage**: encrypted client application states.
- **Peer Client**: the remote party in 1:1 communication.

Then, as shown in Fig. 2, boundaries are defined as:

- **Trusted Client ↔ OS Keyring** Client retrieves or stores root secrets across this boundary. We trust OS keyring for protecting secrets better than ordinary application storage.

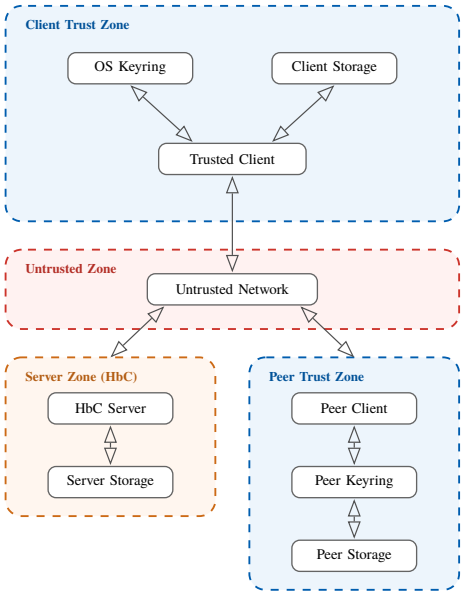

Fig. 2. **Data Flow Diagram** with trust boundaries. Dashed borders delineate trust zones; bidirectional arrows show data flow across boundaries. A detailed walkthrough of the data flow is provided in Section III.B.

- **Trusted Client ↔ Client Storage** Sensitive state must be encrypted before storage because the filesystem itself is not trusted with plaintext secrets.
- **Trusted Client ↔ Untrusted Network** Data leaving client, entering hostile communication environments. TLS ensures client-server transport confidentiality and integrity, while end-to-end encryption ensures client-peer secrecy beyond the server.
- **Untrusted Network ↔ HbC Server** Server receives traffic through hostile network channel. We trust it to follow transport protocols, relay and store data correctly, but do not trust with plaintext message contents.
- **HbC Server ↔ Server Storage** Server requires backend storage for queued ciphertext (short-term) and public identity/prekeys (long-term).
- **Untrusted Network ↔ Peer Client** Messages delivered to the remote endpoint cross the same hostile network environment. Same concerns as above.
- **Peer Client ↔ OS Keyring** The peer client also relies on its OS keyring. Same concerns as above.
- **Peer Client ↔ Client Storage** The peer client also relies on secure local storage. Same concerns as above.

B. Message Data Flow

We detail here the message-delievery data flow assuming that a secure E2EE session has already been established between the two endpoints. For more details on pre-establishment protocols, please refer to Section IV.

The key idea behind this data flow is that plaintext data *never* leaves client runtime; once a message leaves the sender's active application context, it *must* be already encrypted into

our protected protocol data (following X3DH + Double Ratchet protocols).

Step 1: Client Trust Zone

When the sender presses the send button, the **Trusted Client** first loads the conversation's current session state from **Client Storage**. This includes the active ratchet state, message counters, and any key material necessary for setting up the outbound message. Access to these local secrets is mediated by the **OS Keyring**. The message is still plaintext at this point, but it is safe, as it has not yet crossed any untrusted boundary (still within the *Client Trust Zone*).

Client builds outbound packets entirely inside the Client Trust Zone. Afterwhich, ratchet is advanced, fresh message key is derived, plaintext is padded (to reduce length leakage), and payload encrypted. This yields a ciphertext payload plus some limited, non-sensitive delievery-related metadata.

Step 2: Untrusted Zone

Now, the built packet is crossing the **Trusted Client** $\leftrightarrow$ **Untrusted Network** boundary and is being sent to the **HbC Server** over a TLS-protected transport channel.

Step 3: Server Zone

The server can observe messages being sent, its recipient, as well as other routing-related metadata; however, it is virtually impossible for the server to decrypt the packet in any way. The only thing the server can do is relay brainlessly. If the recipient is offline, the server performs store-and-forward by writing the protected packet into **Server Storage**. What is persisted there is the encrypted message body and the minimum metadata needed for later delivery, no recoverable plaintext whatsoever.

Once the recipient reconnects, the server forwards the stored packet back across the **Untrusted Network** toward the **Peer Client** via WebSocket. Upon receipt, the server securely erases the stored packet from **Server Storage** to minimize the risk of *"Harvest now, decrypt later"* attacks.

Step 4: Peer Trust Zone

On the recipient side, the peer client loads its own local session state, verifies that the incoming packet matches the expected session; it also handles any out-of-order or replay-sensitive conditions, which are all capabilities innately supported by Double Ratchet. Then, it derives the corresponding receiving key to decrypt the ciphertext (or load from skipped key database). If verification succeeds, the plaintext is reconstructed and rendered to the user interface, all of which happening inside the peer's trusted runtime.

IV. PROTOCOL DESIGN

Our protocol is a layered design. TLD protects the most outer-layer client-server transprot channel [3], X3DH allows establishment of a shared secret even when recipient is offline [4]. Double Ratchet continuously evolve via a KDF chain (Symmetric-Key Ratchet) to ensure forward secrecy, and resets the shared secret after every round of message exchange (DH Ratchet) to ensure post-compromise security [5]. To further strengthen the security of our protocol, payloads are padded before encryption to reduce exact length leakage, following Signal's design [10]. Any sensitive session states, keys are all sealed at rest using key material derived through HKDF, encrypted via AES-256-GCM-SIV, and guarded by OS keyring [6], [7]. This section details each of these in-depth.

A. Session Establishment

Our session establishment uses **X3DH prekey model** [4].

Prerequisite. Each user device must maintain a long-term identity key (IK), together with a periodically replaced signed prekey (SPK), and a stock of one-time prekeys ($\mathrm{OPK}_{1\ldots N}$). Each of these key comes in as a private + public key pair. For example, $\left(\mathrm{IK}^{\mathrm{priv}}, \mathrm{IK}^{\mathrm{pub}}\right)$ and $\left(\mathrm{OPK}_i^{\mathrm{priv}}, \mathrm{OPK}_i^{\mathrm{pub}}\right)$. Each of these keys serve a unique but important purpose:

- ***Identity Key (*$\mathbf{IK}$*)*** : As the name suggests, this key is binded a specific user device, and represent its long-term identity. This is confirmation the legitmacy of the user's device. Peers can use this as the authoritative ID.
- ***Signed Prekey (*$\mathbf{SPK}$*)***: This key is *signed* by the IK and it allows us to verify the sender's identity without relying on the long-term identity key (which sacrifices forward secrecy). The SPK is periodically replaced.
- ***One-time Prekey (*$\mathbf{OPK_i}$*)***: This key, as the name suggests, is for one-time use only; hence why it comes in a stock. The OPK aims to push forward secrecy to its absolute limit. Instead of relying on periodic resets, we force each X3DH key establishment to be ephemeral.

For a receiver such as Bob, the server stores a *prekey bundle* containing $\mathrm{IK}_B^{\mathrm{pub}}$, $\mathrm{SPK}_B^{\mathrm{pub}}$, a signature over $\mathrm{SPK}_B^{\mathrm{pub}}$ (produced with $\mathrm{IK}_B^{\mathrm{priv}}$), and optionally one available $\mathrm{OPK}_{B,j}^{\mathrm{pub}}$. The corresponding private keys never leaves Bob's client trust zone, as described in Section II.B.

Procedure. When Alice initiates a session with Bob, she starts by fetching Bob's prekey bundle from the server. Before any key agreement, Alice verifies the signature on Bob's signed prekey (SPK) using Bob's identity key (IK). This prevents silent swapping of arbitrary signed prekey that is not actually binded to Bob's trusted identity.

Alice then generates a fresh *ephemeral key* pair $\left(\mathrm{EK}_A^{\mathrm{priv}}, \mathrm{EK}_A^{\mathrm{pub}}\right)$ for this session establishment. Using her own local secrets together with Bob's published prekeys, she computes the X3DH Diffie-Hellman values:

$$\begin{aligned}
\mathrm{DH}_1 &= \mathrm{DH}\left(\mathrm{IK}_A^{\mathrm{priv}}, \mathrm{SPK}_B^{\mathrm{pub}}\right) \\
\mathrm{DH}_2 &= \mathrm{DH}\left(\mathrm{EK}_A^{\mathrm{priv}}, \mathrm{IK}_B^{\mathrm{pub}}\right) \\
\mathrm{DH}_3 &= \mathrm{DH}\left(\mathrm{EK}_A^{\mathrm{priv}}, \mathrm{SPK}_B^{\mathrm{pub}}\right) \\
\mathrm{DH}_4 &= \mathrm{DH}\left(\mathrm{EK}_A^{\mathrm{priv}}, \mathrm{OPK}_{B,j}^{\mathrm{pub}}\right) \text{ (if permitted)}
\end{aligned} \tag{1}$$

These values are concatenated and passed through a KDF to derive the initial shared secret:

$$\mathrm{SK} = \mathrm{KDF}(\mathrm{DH}_1 \parallel \mathrm{DH}_2 \parallel \mathrm{DH}_3 \parallel \mathrm{DH}_4) \tag{2}$$

where DH_4 (the one which uses OPK) is omitted if no one-time prekey was available, for example, due to Bob not being offline for too long and his OPK depleted.

After deriving the shared secret SK, Alice starts messaging Bob. During these initial messages, Alice would attach what we called an *"X3DH Initial Metadata"*, which contains all the information needed for Bob to reconstruct the same secret. This includes, namely, her $\mathrm{IK}_A^{\mathrm{pub}}$, her $\mathrm{EK}^{\mathrm{pub}}$, and information on which signed prekey and one-time prekey were used. The message payload itself is already encrypted using key material derived from SK, so even the very first application message is protected.

When Bob eventually receives these *X3DH Initial Metadata*, he uses his local private keys to recompute the same Diffie-Hellman values from the opposite side (DH properties):

$$\begin{aligned} \mathrm{DH}_1 &= \mathrm{DH}\big(\mathrm{SPK}_B^{\mathrm{priv}}, \mathrm{IK}_A^{\mathrm{pub}}\big) \\ \mathrm{DH}_2 &= \mathrm{DH}\big(\mathrm{IK}_B^{\mathrm{priv}}, \mathrm{EK}_A^{\mathrm{pub}}\big) \\ \mathrm{DH}_3 &= \mathrm{DH}\big(\mathrm{SPK}_B^{\mathrm{priv}}, \mathrm{EK}_A^{\mathrm{pub}}\big) \\ \mathrm{DH}_4 &= \mathrm{DH}\big(\mathrm{OPK}_{B,j}^{\mathrm{priv}}, \mathrm{EK}_A^{\mathrm{pub}}\big) \end{aligned} \tag{3}$$

Thanks to symmetry of Diffie-Hellman, Bob derives the same SK as Alice. At that point, both parties successfully and efficiently (compared to RSA) enter the same initial secure session state without *ever* needing to transport the full, shared secret online.

This design is what gives our system its asynchronous property. Bob does not need to be online during this whole setup process; he only needs to have published a valid prekey bundle in advance. At the same time, the server is limited to distributing public key material and relaying ciphertext. It cannot derive the shared secret unless it compromises the private keys stored only on client devices. Therefore, we successfully meet all requirements by the project, and went a step further.

B. Message Formats

Double Ratchet (DR). For the very first round of encrypted messages in a conversation (before the recipient acknowledge receival), the protected payload is always accompanied by an *X3DH Initial Metadata* described earlier. This includes Alice's identity public key, her ephemeral public key, and identifiers indicating which signed prekey and one-time prekey from Bob's bundle were used. These metadata are necessary because Bob must reconstruct the same initial shared secret from his local private keys.

Then, once a session has been bootstrapped, subsequent messages are *no longer* tie to X3DH metadata. Instead, each encrypted message carries a *Double Ratchet* metadata [5]. At a high level, this metadata contains:

- Sender's current DH ratchet public key
- Current message number within the sending chain
- Number of messages in the previous sending chain

These fields are essential for when messages arrive out-of-order, either in the form of intra-chain out-of-order, or inter-chain out-of-order. Concretely, the current DH public key tells the receiver whether a new DH ratchet step has occurred. The current message number tells the receiver which message key within the active receiving chain should be derived. The previous-chain length allows the receiver to know how many skipped keys from the old chain may still need to be derived and cached when messages arrive out of order.

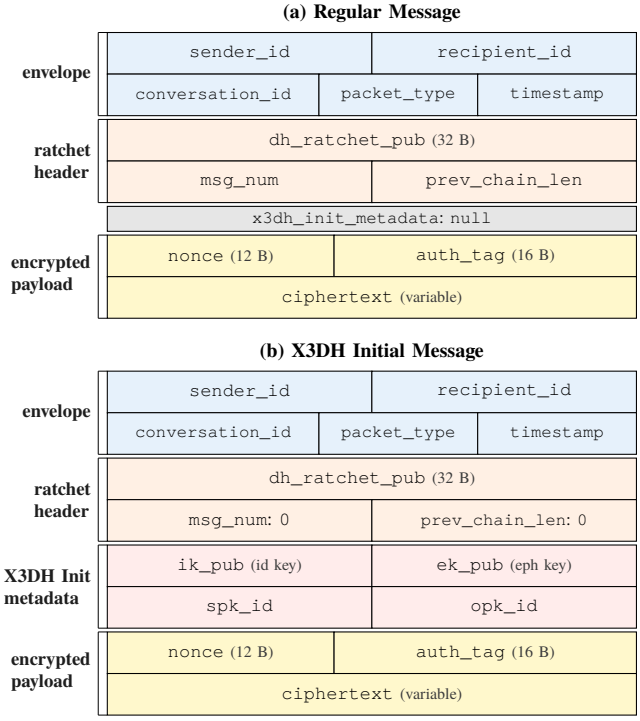

Fig. 3. Packet structure of (a) regular messages and (b) X3DH initial messages, featuring the *X3DH Initial Metadata* discussed in Section IV.

C. State Machine

Our protocol is stateful. Simpler systems may treat each message as an isolated encryption event, but that approach cannot provide the same level of forward secrecy, post-compromise recovery, replay resistance, and asynchronous recovery that modern secure messengers require. In our design, each conversation evolves through a structured state machine whose transitions are driven by X3DH bootstrap events and Double Ratchet updates.

Uninitialized State. Initially, two users may know of each other, but no secure session exists yet (see Fig. 4). In this state, the receiver has only published a valid prekey bundle to the server, while the initiator has not yet derived any shared secret with that peer. The system is therefore ready for session establishment, but not yet ready for ordinary encrypted messaging.

Bootstrap State. Once Alice fetches Bob's prekey bundle and successfully computes the X3DH shared secret, the conversation enters a bootstrap state (Fig. 4). Here, the first root secret is already available locally, but Bob may not yet have reconstructed it. Alice can already send the initial protected message together with the X3DH Initial Metadata, while Bob

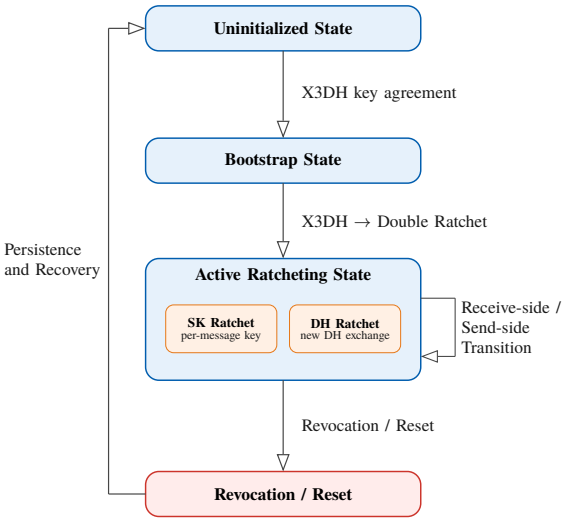

Fig. 4. Session state machine. Rounded boxes are protocol states (matching the state paragraphs below); labeled edges are transitions. The *Active Ratcheting* state contains two continuously operating ratchet mechanisms.

remains one step behind until he receives and processes it. This state is what allows the system to be asynchronous.

Active Ratcheting State. After Bob reconstructs the same shared secret, both parties enter the *Active Ratcheting State* (the central state in Fig. 4). From here on, the X3DH's purpose has been fulfilled and the shared secret is no longer needed. Specifically, the X3DH shared secret is immediately handed to Double Ratchet as the initial root key. From then on, every outbound message advances a *Symmetric-Key Ratchet*, which derives a fresh message key; during each turn of conversation a new DH ratchet public key is exchanged, and the session performs a *DH Ratchet* step, resetting the *Symmetric-Key Ratchet* [5]. This distinction is crucial:

- ***Symmetric-Key Ratchet*** gives us per-message key evolution, so compromising one message key does not expose other message keys in the same chain.
- ***DH Ratchet*** gives us post-compromise recovery, because a fresh Diffie-Hellman exchange injects new entropy into the root secret and cuts off an attacker who only knows an older chain state.

Receive-side Transition. Upon receiving message (the self-loop in Fig. 4), the client first inspects the Double Ratchet metadata. If the ratchet public key is unchanged, the message belongs to the current receiving chain, so the receiver derives keys forward until the stated message number is reached. If the ratchet public key has changed, the client recognizes that the peer has advanced the DH ratchet. It must then finalize the old receiving chain, derive and cache any skipped keys if necessary, mix the new DH output into the root key, and initialize a fresh receiving chain before decrypting the message.

Send-side Transition. When sending, the client advances only its own sending chain unless it has previously received a fresh peer ratchet key. In that case, before sending the next message, it performs its side of the DH ratchet step, updates the root key, initializes a new sending chain, and attaches the new ratchet public key in the message metadata. This alternating process is what allows the ratchet to "take turns" strengthening the session after each round of communication.

Persistence and Recovery. Because the protocol is stateful, the local client must persist session state carefully. This includes the root key, current sending / receiving chain states, message counters, peer ratchet public key, and skipped-message bookkeeping. Losing this state would make future packets undecryptable; leaking this state would weaken session security. Therefore, these values are stored only in encrypted form, protected by locally derived keys backed by the OS keyring [6], [7].

Revocation / Reset. A session may also leave the active state, as shown by the downward transition in Fig. 4. Examples include explicit logout, device removal, local state invalidation, or cryptographic inconsistency detection. In such cases, the session is considered no longer trustworthy, and a fresh X3DH establishment is required before new secure messaging can continue (the left-side return path in Fig. 4). This prevents silent continuation from corrupted or partially inconsistent state.

D. Replay Handling

Under our *Network Attacker* threat model (Section II.A), the adversary is free to capture ciphertext on the wire and reinject it at any later time (i.e. replay attack). Since we cannot trust server with maintaining a stateful sequence filter, replay rejection must be enforced entirely end-to-end and client-side. We identified two types of replay attack that can occur:

1) *Message-Layer Replay*: adversary duplicates and replays captured Double Ratchet ciphertext.
2) *Session-Bootstrap Replay*: adversary duplicates and replays captured X3DH init-bearing packet in an attempt to reset or desynchronize the session, potentially resulting in DoS

We detail both below.

1) Message-Layer Replay. Fortunately, Double Ratchet is inherently immune to this type of attack. Replay at this layer is *structurally* rejected by the Double Ratchet algorithm itself. This is because each message key MK_i is derived from a forward-only KDF chain: each key is derived by hasing the previous state; therefore, if the receiving KDF chain advanced past position i, the same MK_i can no longer be recomputed from current state. Therefore, duplicated old message **cannot** be decrypted, even if the client wants to. We call this the *one-time-use invariant* of Double Ratchet.

Out-of-order duplicates are handled by the same one-time-use invariant, but applied to skipped message keys. Recall that we sometimes need to skip ahead in the KDF chain if later packets arrived before older packets, necessitating temporary storage of skipped message keys. This does not, however, affect our replay protection, because the mechanism is invariant to order. Concretely, skipped keys are stored under $(\mathrm{dh}_{\mathrm{pub}}, n)$ and consumed via a *destructive* `state.skipped.pop(...)` operation, so a second lookup for the same key returns nothing, structually preventing

decryption. The ratchet header is also bound into the AEAD associated data, which prevents an attacker from silently relabelling a replayed ciphertext to a fresh chain position without breaking AEAD verification.

We further bound the storage parameters of skipped keys ($S_{\max} = 1000$ per chain advance, $C_{\max} = 5$ retained chains, 7-day TTL on skipped keys; see Table I) to ensure this storage cannot grow unboundedly in case of adversarial skip patterns (attacker intentionally causing skip, in an attempt to cause DoS).

TL;DR: Replay protection at this layer is not implemented as a filter at all. **It is resistant by construction**: it is a direct consequence of one-time-key consumption plus AEAD-authenticated headers.

2) X3DH init replay. Replay at this session-bootstrap layer is more subtle and deserves a closer look compared to Double Ratchet. By design, our sender re-attaches the X3DH Initial Metadata to *every* outbound message until the peer has proven receipt of a session key (persisted locally as `UnackedX3DHInitRow`). This is necessary because the very first init-bearing packet may be lost in transit, and if that occurs, the recipient will enver be able to recover the session. It is possible that the attacker can capture such an init packet and replay it multiple times, which will constantly reset the session if the recipient naively accepts every init as a fresh session, causing DoS.

A naive but tempting approach is to keep track of a set of every init tuples it has ever processed, and reject any packet whose tuple is already in the set. This indeed kill replay, but also kills our aforementioned design (allowing the same init metadata to be repeated until the peer acknowledges).

Therefore, we instead enforce this through **One-time Prekey (OTP) Single Use.** The server atomically deletes an OPK once someone claims it. From the recipient client's perspective, it destructively consumes the matching OPK stated in the sender's X3DH init metadata. Therefore, after first use, it is cryptographically infeasible to ever reconstruct the same X3DH shared secret anymore, even if the attacker replays the same packet with the same X3DH init metadata because: a) the server no longer holds the OPK^{pub} and b) the client no longer holds OPK^{priv}.

V. CRYPTOGRAPHIC CHOICES

A. Primitives

- **X25519.** For Diffie-Hellman end-to-end key agreement.
- **Ed25519.** Digital-signature primitive for X25519 keys.
- **HKDF-SHA256.**
  1) To derive X3DH shared secret from Equation (2).
  2) To evolve Double Ratchet RK after each DH step.
  3) To utilize AEAD encryption key plus nonce.
  4) To derive per-purpose local-encryption keys from a single OS-keyring-backed root secret [7].
- **HMAC-SHA256.** KDF primitive for DR chains.
- **AES-256-GCM-SIV.** Default encryption primitive used widely. GCM-SIV variant for nouce-misue resistance compared to ordinary GCM. [6].
- **Argon2id.** Main primitive for server-side password-hashing. Chosen due to being memory-hard [8].
- **TOTP.** For multi-factor authentication, instantiated with **HMAC-SHA1** exactly as in the TOTP standard and the Python `cryptography` implementation [9].
- **SHA-256.** Though not part of E2EE itself, SHA-256 is still used in various areas such as fingerprint.
- **CSPRNG.** Our implementation relies on OS-backed CSPRNG output for X3DH/ratchet key generation, Argon2 salts, TOTP secrets, bearer tokens, and local AEAD nonces. All other primitive above assumes high-quality randomness at key-generation time.

As mentioned before, in addition to primitives, we incorporated two higher-level cryptographic constructions: **X3DH** for asynchronous session establishment and **Double Ratchet** for continuous key evolution [4], [5]. We additionally use **associated data** to authenticate protocol metadata without encrypting it, and **ISO 7816-4 style 0x80-then-zero padding** [10]. These are not separate primitives in the same sense as X25519 or AES-GCM-SIV, but they are nevertheless cryptographically meaningful parts of the final design.

B. Libraries

We use two security-related libraries:

- The `cryptography` library[2].
- The `keyring` library[3].

Our cryptographic implementation relies entirely on the Python `cryptography` library; it provides us secure and reliable implementations of Ed25519, X25519, HKDF, HMAC, AES-GCM-SIV, Argon2id, and TOTP used across our client and server. The python `cryptography` library is amonst one of the most well-reviewed and widely-used cryptographic libraries in Python. Therefore, this choice is a directly result of our trust assumption of *Secure Cryptographic Primitives*.

Local secret sealing too relies on `cryptography` library, along with the Python **`keyring`** library. `keyring` stores the 32-byte root secret inside an OS-native secure backend (e.g., MacOS Keychain), whereas the actual encryption and decryption are performed by our own code through `cryptography`, using AES-256-GCM-SIV and HKDF-derived purpose keys.

We also explicitly reject insecure keyring backends. This is a subtle but important design choice: a system that encrypts local state but stores its root key in plaintext is merely an illusion of security. Our implementation fail-fast when no secure OS-backed keyring exists. This is an explicit design choice, not a bug—DoS is better than silent insecurity.

Also, it is worth noting that, for Argon2, we used the **OWASP recommended parameters** to make stolen password

[2]https://github.com/pyca/cryptography

[3]https://github.com/jaraco/keyring

TABLE I
CRYPTOGRAPHIC PARAMETERS.

Parameter	Value
Authentication	
Argon2id salt s	16B
Argon2id hash length h	32B
Argon2id memory m	47 104 KiB [8]
TOTP secret length K	20B
TOTP digits d	6
TOTP time step X	30 s
TOTP window w	$\pm$1 step
TOTP hash H	HMAC-SHA-1 [9]
Bearer token length n_{tok}	32B
Token store hash	SHA-256
Key Agreement (X3DH)	
Ed25519 public key pk_E	32B
X25519 public key pk_X	32B
Bundle signature sig_E	64B
KDF hash H	SHA-256
KDF salt	32 bytes of `0x00`
KDF info	`VERACITY-X3DH`
KDF padding F	32 bytes of `0xff`
Double Ratchet	
Root KDF hash H	SHA-256
Root KDF output L	64B $\rightarrow$ 32 B root + 32 B chain
Chain KDF msg-key constant c_{mk}	`0x01`
Chain KDF chain-key constant c_{ck}	`0x02`
Msg-key expand hash H	SHA-256
Msg-key expand output L	44B $\rightarrow$ 32B AES + 12B nonce
Header size	40 B
Max skipped messages S_{max}	1 000
Max stored chains C_{max}	5
Skipped key TTL	7 d
Padding block B_{pad}	80B (following ISO 7816-4 [10])
Storage & Prekeys	
At-rest root key K_{root}	32B
At-rest purpose key K_p	32B
At-rest nonce n	12B
Sealed format	`v2:`$\langle$base64$\rangle$
Signed prekey lifetime T_{SPK}	30 day
OPK pool size N_{OPK}	100

hashes materially more expensive to brute-force, while keeping registration and login still practical for the server [11].

C. Versions

At dependency level:

- Client & server require `cryptography >= 46.0.5`.
- Client requires `keyring >= 25.6.0`.

These are all detailed in the `pyproject.toml` files in the codebases, in both `server/` and `client/`.

At protocol level: client-server transport uses TLS1.3 [3].

VI. SECURITY ANALYSIS

In this section, we analyze the security of our system. Specifically, we analyze it against each of the threat models described in Section II.A, and show that our system are resistant to all said adversaries.

A. Honest-but-Curious (HbC) Server

Under HbC model, the server attempts to sniff as much information as possible from the messages. Therefore, this threat model primarily concerns successful end-to-end encryption. Our system is resistant to such attack thanks to X3DH and Double Ratchet:

E2EE encryption. E2EE is automatically guaranteed in any systems where the two endpoints hold a shared secret. This is definitely achieved in your setup, and in a much more secure manner compared to traditional key transport.

B. Network Attacker

Given that our message packets are end-to-end encrypted, it is impossible for network attackers to decrypt right away. Therefore, the main concerns are *Harvest now, decrypt later* attack, post-compromise surveillance, and replay attack.

Forward Secrecy (*Harvest Now, Decrypt Later*). This type of attack can actually occur in two manners: 1) harvest the secret exchange, and 2) harvest in-transit message packets. By capturing in-transit message packets, attackers can wait until they obtain the shared secret key to decrypt the message; by capturing the secret exchange, the attacker can obtain the secret key later, and in turn decrypt captured messages.

- 1) *Secret Exchange Harvest*: **X3DH** addresses this. Traditional key transport approach *cannot* defend against this, as it transmit the entire shared secret over the network, which is inevitably harvestable. Our X3DH key agreement algorithm, on the other hand, is naturally resistant to this attack, as it *does not* share secrets over the network. X3DH uses the Diffie-Hellman (DH) algorithm, which allows both party to converge onto the same shared secret simply by sharing two temporary public keys. The private part of the keys are erased once X3DH completes.
- 2) *Message Packet Harvest*: **Double Ratchet**'s Symmetric-Key Ratchet addresses this. Traditional static key approach risks exposing all messages once this static shared key got exposed. In Double Ratchet, a symmetric-key ratchet constantly derives new shared secrets by repeatedly hashing previous states, creating a KDF chain. Once a new state is derived, previous states are securely erased. This makes decrypting harvested message packets impossible, as the keys needed to decrypt have been deleted by the user.

Post-Compromise Security. Forward secrecy prevents the malicious attacker from decrypting old messages, but the attacker might still be able monitor all future messages. Consider the aforementioned symmetric-key ratchet: hash functions are one-way, making it impossible for attackers to recover old message keys; however, attackers can derive *all* future secret states from now on (i.e., no post-compromise security). **Double Ratchet**'s Diffie-Hellman Ratchet addresses this. The DH Ratchet resets the KDF chain after every turn of message exchange. Utilizing RSA intuition, during each message turn, both parties exchange a new pair of g^a and g^b,

which they then computes a shared g^{ab}. Therefore, in case the shared secret ever got leaked, malicious actors (e.g., the curious server) can only gain access to a limited window of plaintext message, which then quickly closes after a single turn of conversation due to KDF chain reset.

C. Attempt Unauthorized Access

As discussed earlier, we assume that attacker will attempt password guessing, OTP guessing, credential, and bearer-token theft. *Veracity* address each of this carefully:

1) *Password Guessing*: **Argon2id** password hashing and salting addresses this. Hashing already makes rainbow-table attacks hard and database leaks safe. Argon2id takes this a step further by being memory-hard, thus making it more resistant to GPU and ASIC processors.
2) *OTP Guessing*: **Time-based one-time code** addresses this by resetting every 30 seconeds [9]. Compared to other approaches, such as email-based OTP, which usually permits a 5-15 minute time window, TOTP is much stricter and thus resistant to brute-force guessing.
3) *Token Theft*: **Token revocation and expiry** addresses this by narrowing attacker's window of opportunity.

D. Partial Client-side Compromise

We accounted for two types of client-side adversaries: 1) an adversary presented on the user's device (e.g., malware) but has *limited* system privilege, and 2) an adversary that was able to *temporarily* gain access to the user's current cryptographic states or *permanent* access to previous states.

1) *Malware (w/ Limited Privilage)*: **OS Keyring** addresses this by protecting local storage
2) *Secret State Access*: As mentioned in Section VI.B, we address this through the post-compromise security and forward secrecy provided by X3DH and Double Ratchet [4], [5].

VII. TESTING

To demonstrate that *Veracity* works as a complete secure messaging system, perform an end-to-end live workflow with two users. The demonstration was organized into four phases corresponding to the major system capabilities:

1) Account registration and login with TOTP.
2) Friend request showing working X3DH key agreement and session establishment
3) Messaging showing working Double Ratchet and offline queueing
4) user-facing security behavior/feature such as unread counters, delivery state, timed self-destruct, and key verification / key-change warning handling.

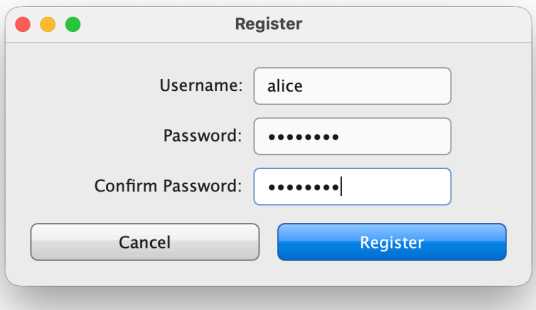

Fig. 5. The registration dialog, asking for the user (@alice) to register an account using an unique username, password, and password confirmation.

Fig. 6. The user (@alice) is asked to enter the TOTP before she complete the registration workflow. Any user need to scan the QR code to add the TOTP provisioning URI to their password manager (for example, 1Password). Then they will enter the TOTP for the first time to finish the account registration.

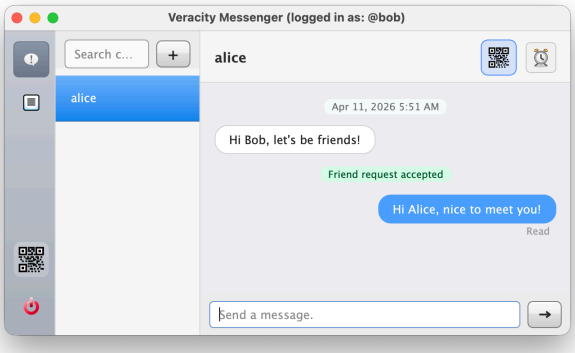

Fig. 7. @alice is trying to add @bob as friend. @bob accepted the friend request and sent the first regular message to @alice. Note that @alice is online and has read the message.

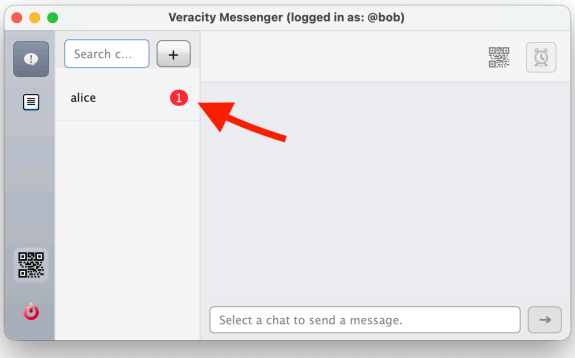

Fig. 10. On @bob's side, he is just back, and he has not open @alice's chat yet. There will be an unread counter indicating @alice sent a message to him and he has not read it yet.

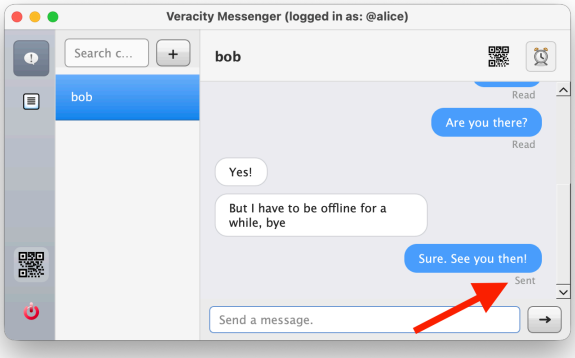

Fig. 8. After @alice and @bob has chatted for a while, @bob has to be offline for a while. Any subsequent messages sent by @alice will be marked as 'sent', indicating these messages are successfully sent to the server, but are waiting in the queue. These messages will be delivered to @bob once he goes back online.

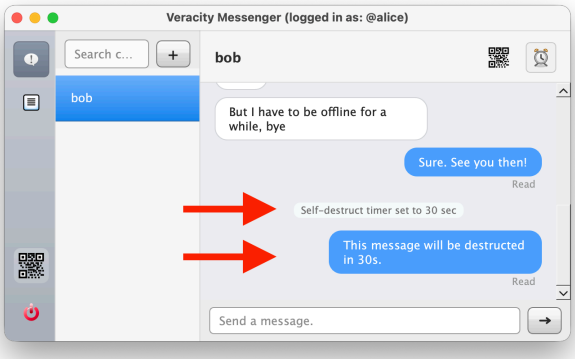

Fig. 11. @alice or @bob updated the message destruct timer to 30 seconds. Any subsequent messages will adopt this destruct timer option and destruct themselves in 30 seconds.

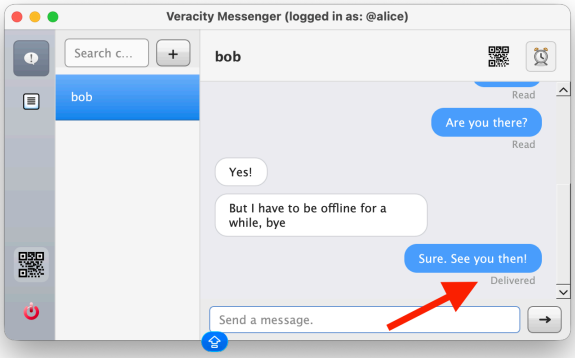

Fig. 9. @bob just went online. So @alice's message will be delivered to @bob. That message status changes to 'delivered', indicating that @bob has received the message (but has not read it yet).

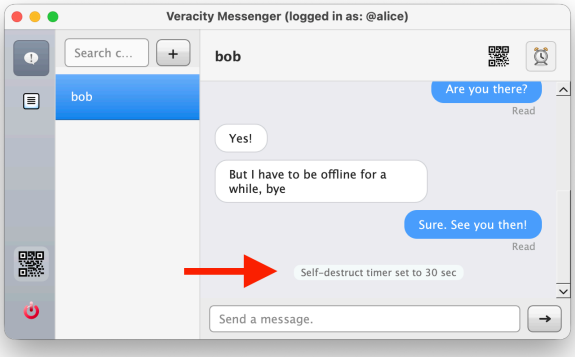

Fig. 12. The message has been destructed after 30 seconds.

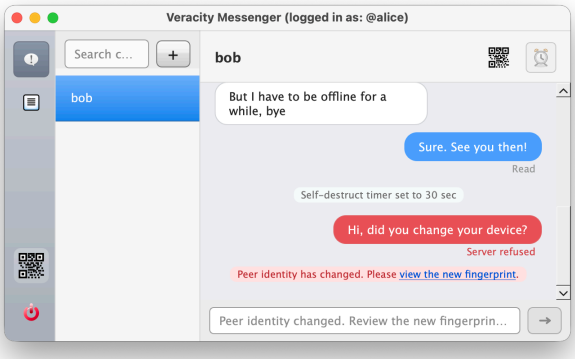

Fig. 13. @bob changed the device, so his fingerprint changes. Now @alice is trying to send a message to him, but the message is rejected. Unless @alice checks @bob's new fingerprint and trust it, @alice cannot send more message to @bob.

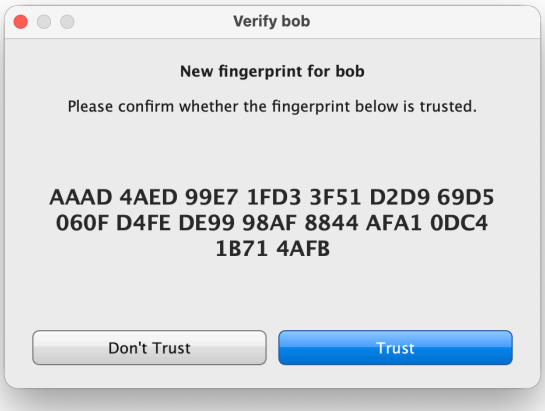

Fig. 14. Now @alice is viewing @bob's new fingerprint. She can verify this fingerprint with @bob via other methods (for example, they can meet in person). @alice can choose to trust, or not to trust @bob's new identity. If she doesn't trust it, she can't send more message to @bob. On the other hand, if she trust it, her client will update @bob's new identity and notify the server, and she can send more messages to @bob. Note that if @bob has change the device and lost all contact data, @alice has to send another friend request before sending any regular messages.

The demonstration starts with two newly registered users. Both completing password + OTP login. *Veracity* client then generates device identity material locally and uploaded the required prekey bundle, including signed prekeys and one-time prekeys. *Veracity* uses industry leading technology and depends on device-bound identity and asynchronous prekey material rather than on a shared secret.

Contact establishment and first secure session. After account and device setup, one user sent a friend request and the other accepted it. We then initiated the first secure conversation as the demo exercised the X3DH-backed bootstrap path. This phase verifies that the system can move from zero shared state to an authenticated secure session using the normal application workflow.

Online messaging, offline queueing, and later delivery. We next exchanged encrypted messages while both users were online. To demonstrate asynchronous messaging, one user was then disconnected and the sender continued submitting encrypted message packets. These ciphertext packets were queued by the server and delivered after the recipient reconnected. This phase demonstrates that the deployed system supports server-mediated offline messaging without requiring the server to terminate end-to-end protection.

VIII. EVALUATION

A. Overview

Our evaluation covers the two most significant trust boundaries of *Veracity*: **Client-Server** and **Client-Client** boundary:

- **Client-Server Branch**: evaluated through bounded formal verification using Temporal Logic of Actions+ (TLA+) and Alloy [1], [2].
- **Client-Client Branch**: evaluated through MITM interception of data and conduct statistical analysis using the statistical test suite NIST 800-22 [12].

Collectively, they create a unified and coherent evaluation framework: server-side lifecycle correctness guaranteed by formal methods, and empirically verified peer-to-peer confidentiality by MITM + SP 800-22.

B. Client-Server Branch

TABLE II
FORMAL VERIFICATION COVERAGE

Object	Property (*e.g., Invariants*)	Tool
Session	accept ⇒ live-only	TLA+
Session	revoke/expire absorb	TLA+
Login	success ⇒ fresh sess	TLA+
Login	failure ⇒ no sess	TLA+
Rate limit	in-proc bounded	TLA+
Act. device	≤1 per user	Both
Curr. SPK	≤1 per device	Alloy
OPK	single-use	TLA+
OPK	claim oldest-first	TLA+
Bundle	serve only if active+SPK	TLA+
Dev. route	needs bound active sess	TLA+
Activation	success ⇒ revoke siblings	TLA+

The Client-Server branch evaluates the server-side subsystems that concerns authentication, session validity, device activation, and key-distribution decisions. This is *Server Zone* discussed in Section II.B that all clients rely on before secure messaging can even begin.

Question. Does the server-side auth, session, and devicekey workflow preserve its intended safety properties across bounded executions?

Method. We implemented various TLA+ modules checked with TLC (the model checker of TLA+) as well as an Alloy model complementing them by check structural constraint (e.g. only one current signed prekey per device). A complete coverage summary is detailed in Table II.

Result. TLC completed successfully on all TLA+ modules. Alloy model also completed successfully on the structural constraint model. Therefore, it can be confidently concluded that, within the given bounded state space, no security invariants were being violated.

Interpretation. Our Client-Server branch evaluation results supports a clean claim: the server-side logic of *Veracity* is implemented securely given our explored scope. Concretely, this means that the evaluated auth, session, device, and prekey transitions all behaved consistently and meeting intended security rules. Therefore, the server-side control plane behaves safely.

C. *Client-Client Branch*

The Client-Client branch evaluates the end-to-end confidentiality goal of the system. This is the central purpose of the entire project, and should thus be taken most seriously. Although messages are relayed by the server, the security claim is ultimate about the two peers: are message content remained protected client-to-client, even under hostile network environments?

Question.

1) Can our client-to-client encryption withstand Man-in-the-Middle (MITM) attack, and
2) Do the ciphertext in-transit contain any detectable statistical fingerprint? Or equivalently, are the message payload indistinguishable from noise?

Method.

1) We intercept and perform MITM attack real traffic via `mitmdump` and `mitmweb`. Specifically, we intentionally configure the client to route through a local MITM certificate via `SSL_CERT_FILE`; by doing so, we strip away any TLS encryption confounders.
2) This therefore allows for running the NIST SP 800-22 Statistical Test Suite (STS) on the exposed message payload to identify the subtlest statistical fingerprints. Specifically, the following tests were performed:
   - **Freq:** Frequency (Monobit) Test
   - **BlkFreq:** Frequency Test within a Block
   - **Runs:** The Runs Test
   - **LRuns:** Longest-Run-of-Ones in a Block
   - **Rank:** Binary Matrix Rank Test
   - **FFT:** Discrete Fourier Transform (Spectral) Test
   - **NonOL:** Non-overlapping Template Matching
   - **OL:** Overlapping Template Matching Test
   - **Universal:** Maurer's *Universal Statistical* Test
   - **Linear:** Linear Complexity Test
   - **Serial:** Serial Test
   - **Entropy:** Approximate Entropy Test
   - **Cumsum:** The Cumulative Sums (Cusums) Test
   - **RandExc:** Random Excursions Test
   - **RandExcVar:** Random Excursions Variant Test.

Results.

1) The traffic review analyzed 199 events (66 HTTP and 133 WebSocket events). Results reveal that *no* plaintext chat message were observed in the relayed flow, other than some insensitive metadatas.
2) TABLE III
   NIST RESULTS FOR 104 SEQUENCES OF 10^6 BITS EACH ($\alpha = 0.01$).

Suite	N	Thresh.	Pass	p-value	Verdict
Freq	104	99	102	0.384	PASS
BlkFreq	104	99	104	0.276	PASS
Runs	104	99	103	0.575	PASS
LRuns	104	99	103	0.978	PASS
Rank	104	99	102	0.237	PASS
FFT	104	99	103	0.798	PASS
NonOL	104	99	100–104	1e-4–0.99	PASS
OL	104	99	104	0.049	PASS
Universal	104	99	104	0.182	PASS
Entropy	104	99	104	0.679	PASS
Serial	104	99	103–104	0.33–0.76	PASS
Linear	104	99	103	0.851	PASS
Cumsum	104	99	101–102	0.46–0.96	PASS
RandExc	67	63	65–67	0.11–0.85	PASS
RandExcVar	67	63	64–67	0.01–0.90	PASS

We further performed rigorous statistical test using the NIST 800-22 STS. Over one hundred 10^6 bit long sequences were captured, totally over 1.04×10^8 bits analyzed. As shown by Table III, *all* statistical test suites were passed, indicating that the the MITM intercepted data payloads does *not* contain any detectable statistical fingerprints.

IX. FUTURE WORKS

In this project, we pushed beyond the project baseline requirements and implemented 1) **Advanced cryptographic algorithms** such as X3DH and Double Ratchet, 2) **Rigorous evaluation** by utilizing formal verifications (TLA+ and Alloy) as well as MITM interception of data payloads and analyzing them with the NIST 800-22 Statistical Test Suite [12]. However, several limitations and future work requires discussion.

Post-Quantum Security. While our system implements advanced algorithms (X3DH and Double Ratchet) that offers the highest attainable level of security assuming the hardness of the Elliptic Curve Diffie-Hellman problem (ECDH) under classical computational models, we did not account for *post-quantum security*. It is unlikely, though not impossible that, in the far future, quantum computing technology may advance to the point where even our current design can be compromised. One future work direction is to replace X3DH and Double Ratchet with *Post-Quantum Extended Diffie-Hellman* (**PQXDH**) [13] and **Triple Ratchet** which run a Double Ratchet and a Sparse Post-Quantum Ratchet in parallel [5].

Multi-Device Synchronization. Our X3DH key agreement mainly concerns the asynchronous key agreement between two endpoints. However, the recipient user may have multiple devices, necessitating simultaneous initialization with all recipient devices; furthermore, both parties may undergo device deletion. X3DH is insufficient for this, and a future direction may be to implement Sesame algorithm [14].

X. REQUIREMENTS CHECKLIST

(R1) Registration

1) Users can register an account with a unique username. The server enforces this uniqueness. During registration, it first checks if the username existed. If so, a conflict error is returned.
2) User passwords are not stored in plain text. Instead, they are hashed with Argon2id (based on OWASP recommened parameters [11]). Each user gets a randomly generated salt, which is included and stored in the PHC-formatted hash string.
3) A basic password rule is required. Passwords must be 8-128 characters long and include at least one letter, one number, and one symbol.
4) Both registration and login are rate-limited. The server currently uses an in-process fixed-window rate limiting method. Both registration and login are limited to 5 requests per 300 seconds. The rate limit key is based on `(username, client_ip)`, while registration currently uses only `client_ip`. If the limit is exceeded, the server rejects further requests and returns a *429 Too Many Requests* error.
5) In addition, during registration, TOTP setup is mandatory. An extra verification step is in-place to prevent users from forgetting to save their OTP provisioning URI (for example, in a password manager), which could cause denial of service. For details about the login process and OTP-related flows, please refer to R2.

(R2) Login with Password + OTP

1) Users need to provide their username, password, and a 6-digit TOTP code to log in.
2) The server creates a new session for each successful login, along with an access token returned to the user. Afterwards, all user requests requires an embedded access token. Upon log out, the session access token is revoked.
3) The server does not store the raw access token, but a SHA-256 hash of the token. The server verify the correctness of the token by hashing it and comparing it against the database. This mechanism ensures that even if the database is leaked, tokens are secretive.
4) A session is linked to a specific user device, and a token can only represent its creator. Messaging-related endpoints will require session information to identify the user, and a token.

(R3) Logout / session invalidation

1) The user can choose to log out of their account. The server will be asked to revoke this session and invalidate the access token. Any further requests using this token will be rejected.
2) The server will also proactively check if a session is outdated. There is an `expires_at` field in each session, and the server will use this field to check if the session expires when a request comes.

(R4) Per-device identity keypair

1) Each user device will generate **two** long-term identity keypairs: an Ed25519 signing keypair and an X25519 keypair for the actual X3DH key agreement.
2) In addition to identity keys, user devices further generate and maintain a periodically updated Signed-prekey (SPK) and a bucket of One-time prekeys (OPK).
3) The clients will upload the corresponding public materials to the server. These public materials are necessary for asynchronous initialization of X3DH sessions. In other words, it allows a user to request another user's prekey bundle and derive a shared secret, even if the recipient is *not* online.
4) The corresponding private materials are stored in the client's local base. These materials never leave the client's trusted zones (see Section II.B). These private materials are encrypted at-rest.

(R5) Fingerprint / verification UI

1) In our implementation, the client provides a fingerprint for each contact (i.e., device identity), including the logged-in user itself. The fingerprint is computed based on `device_id`, device identity signing public key, and device identity Diffle-Hellman public key.
2) The UI allows users to view both their own fingerprint and a contact's fingerprint, and also provides a QR code to support out-of-band verification, for example, they can meet in person to verify each other's fingerprint.

(R6) Key change detection

1) The client will record each contact's last-trusted device identity. The server maintains a mapping that records which device of a recipient is trusted by a given sender.
2) When the recipient's identity changes (this may happen when they change their device or reset their local database), the server rejects to forward messages and will notify the sender. The sender will receive a UI dialog showing the recipient's new fingerprint and asking the user whether to trust it. Before trusting the new fingerprint, no message are allowed to send.
3) Once marked trusted, the new identity is recorded as the 'latest trusted identity' and the previous identity will be obsolete. The server will also update the corresponding mapping records. Finally, a new X3DH session will be initialized.
4) Note that we have realized that the server may be compromised. However, both X3DH and Double Ratchet protect the end-to-end messages cannot be exposed to server, so we only need to consider the peer's identity. The mechanisms described above are just designed to defend against a compromised or malicious peer.

(R7) Secure session establishment

1) We use X3DH (Extended Triple Diffie-Hellman) for shared secret agreement.
2) Once a shared secret is bootstrapped via X3DH, subsequent secret derivation are delegated to Double Ratchet. Specifically, the shared secret acts as the initial root key for the KDF chain generated by Double Ratchet.

3) The details of X3DH and the Double Ratchet are described in earlier sections.

(R8) Message encryption and authentication

1) Message content is protected by the Double Ratchet, which derives a *unique* message key for *every* single message. These keys are then used by AES-256-GCM-SIV for encryption and authentication. We chose AES-256-GCM-SIV for two reasons:
   a) It prevents nounce-misuse, which adds an extra layer of security in-depth.
   b) It offers AEAD, which ensures associated data (AD) integrity. Any modification to either the ciphertext or the AD will fail-fast.
2) Before encryption, the plaintext is padded using ISO/IEC 7816-4 padding, which significantly reduces the risk of information leakage through message length.

(R9) Replay protection / de-duplication As mentioned in Section IV.D, the Double Ratchet algorithm is naturally replay-robust. Deduplication is also implicitly handled by Double Ratchet as secret states update after every message; thus old, duplicated messages cannot be decrypted, be it benign or malicious.

(R10) TTL / expiration policy

1) Each message includes a `ttl` and an `expires_at` field, and once the `expires_at` time is reached, the message is immediately deleted from both the UI and the database.
2) After deleting a message, the client immediately looks for the next message that needs to be deleted and sets a timer to remove it when the time is reached.
3) If either party updates the destruct timer, then any messages sent afterward by either side will use the new `ttl` value, and the `expires_at` field will be calculated based on the send time; both sides will delete these messages when their `expires_at` time is reached.

(R11) Client deletion behavior

1) Please refer to R11.
2) Deletion on the client happens simultaneously in both the UI and the database, so messages to be deleted are removed from memory and disk almost instantly, with virtually no delay in between.

(R12) Server storage behavior (best-effort) The server does not store any messages except those that must be held when the recipient is offline.

(R13) Friend request workflow User can send friend request by the recipient's username. We designed a complete friend request workflow and even considered various edge cases, such as whether the users are already contacts or are blocked. Specifically, we considered the following cases:

1) The friend request sender can send friend requests to others. If the recipient is not in the sender's contact list, the first message will automatically be a friend request message.
2) If the two parties are not yet contacts, after sending a friend request, the sender can choose to cancel it, while the recipient can choose to accept, decline, or block the sender. Before the recipient accepts or declines, or the sender cancels, neither party can send any other messages.
3) If the recipient is already a contact of the sender, both sides will receive the same prompt as stated in 2). If the recipient declines the friend request, they will remove the sender from contacts.
4) If the sender has blocked the recipient, they cannot send a friend request to them.
5) If the recipient has blocked the sender, the sender's friend request will be discarded immediately. Any attempts to send or cancel the request will not be visible to the recipient, and the sender will not know they have been blocked, though they may wonder why their request never receives a response.
6) The sender's messages may be rejected, so they can try sending a friend request to reestablish the contact relationship, even if the recipient is in their contact list.
7) If the recipient's device identity changes for some reason, the sender may need to reestablish the friendship, but the sender might not be able to create it again. This is a necessary trade-off made for security reasons.
8) A user is not allowed to send a friend request to themselves.

(R14) Request lifecycle Please refer to R13.

(R15) Blocking / removing

1) Messages from blocked users are not received and are instead discarded.
2) The sender is notified that their message was rejected, but they are not informed whether they were removed as a contact or blocked.
3) The sender may try to send a friend request. However, if the recipient has blocked the sender, the friend request will also be discarded. Please refer to R13.

(R16) Default anti-spam control Before the friend request is accepted, the sender can send only one message.

1) If there is a pending state, the sender is not allowed to send more messages.
2) Even if additional messages are sent, the receiver will discard them.

(R17) Minimum delivery states

1) A message is considered sent once it reaches the server. Messages that are queued on the server (because the recipient is offline) or can be delivered directly to the recipient are both considered sent.
2) If the message does not reach the server (for example, due to a network issue between the sender and the server), it will be marked as send failed.
3) Only messages that are actually delivered to the recipient and acknowledged with a `DeliveryAckEnvelope` will be marked as delivered.

4) Messages that are rejected by the server due to a change in the recipient's device identity that the sender has not trusted will be marked as server refused.
5) Friend requests do not display a message status.

(R18) Define "Delivered" semantics Please refer to R17.

(R19) Metadata disclosure statement
1) Friend requests, regular messages, delivery acks, read acks, and destruct timer updates are all wrapped in a unified `Envelope` data structure, and the `Envelope` as a whole is encrypted by the Double Ratchet, so the server cannot see the actual content or meaning of the messages.
2) The server can only access some necessary metadata, such as the usernames of the sender and recipient, whether the recipient is currently online, whether the sender's message should be queued for delivery when the recipient comes online, and whether the recipient has come back online and is attempting to fetch messages from the queue.
3) Based on timing, the server may also be able to infer some information about delivery and read acks, but this is unavoidable.

(R20) Offline ciphertext queue
1) When the sender sends a message to the recipient, the server first checks whether the recipient is online.
2) If the recipient is online, the message can be pushed directly to them via WebSocket, and the server neither stores nor queues the message.
3) If the recipient is offline, the server queues the message payload (which has already been encrypted on the sender's side and cannot be read by the server) and delivers it when the recipient comes online.
4) When the recipient comes online and connects to `/ws/packets`, the server first sends all queued offline packets in order, and then marks that connection as an active online connection for the recipient.
5) Queued packets are immediately deleted once received by the recipient. The server performs a secure erase of the queued packets via the `PRAGMA secure_delete` and `PRAGMA journal_mode = DELETE` flag in SQLite. Without this, the database only marks queued packets as deletion but never actually erase it from the disk, thus increasing the attack surface of *Harvest now, Decrypt later* attacks.
6) Each offline packet should record which active device it is intended for. If the recipient changes devices (i.e., the active device identity changes), these packets will be discarded without notifying either the sender or the recipient.

(R21) Retention and cleanup
1) An uncompromised server does not retain messages that can be delivered immediately. It only stores messages that must wait for the recipient to come online.
2) The server has no knowledge of the message payload, so it cannot know whether the message has a self-destruct timer. All logic related to message self-destruction is handled entirely on the sender and recipient clients.

(R22) Duplicate/replay robustness As mentioned in Section IV.D, the Double Ratchet algorithm is naturally replay-robust. Deduplication is also implicitly handled by Double Ratchet as secret states update after every message; thus old, duplicated messages cannot be decrypted, be it benign or malicious.

(R23) Conversation list The main window will display all recent contacts, regardless of whether a friendship has been established, along with their corresponding conversations.

(R24) Unread counters
1) Messages that are delivered but not yet read (i.e., no `ReadAckEnvelope` has been sent back to sender) will be counted.
2) A sender's friend request message is counted as an unread message as long as it has not been accepted, rejected, or blocked.

(R25) Paging / incremental loading
1) Our UI and database both use pagination. Each request retrieves only `CHAT_PAGE_SIZE` messages at a time, with a default value of 50.
2) When the user needs to load more messages (i.e., they have scrolled to the top of the chat), a "Load more messages" button is shown. Clicking this button fetches and displays an earlier set of `CHAT_PAGE_SIZE` messages from the database.

References

[1] M. A. Kuppe, L. Lamport, and D. Ricketts, "The TLA+ toolbox," *arXiv preprint arXiv:1912.10633*, 2019.
[2] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
[3] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. doi: 10.17487/RFC8446.
[4] M. Marlinspike and T. Perrin, "The X3DH Key Agreement Protocol." Nov. 2016.
[5] T. Perrin, M. Marlinspike, and R. Schmidt, "The Double Ratchet Algorithm." Nov. 2025.
[6] S. Gueron, A. Langley, and Y. Lindell, "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption," RFC 8452, Apr. 2019. doi: 10.17487/RFC8452.
[7] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, May 2010. doi: 10.17487/RFC5869.
[8] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications," RFC 9106, Sept. 2021. doi: 10.17487/RFC9106.
[9] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "TOTP: Time-Based One-Time Password Algorithm," RFC 6238, May 2011. doi: 10.17487/RFC6238.
[10] Signal Messenger, LLC, "Data+MessagePadding.swift." 2022.
[11] OWASP Cheat Sheet Series Team, "Password Storage Cheat Sheet."
[12] A. Rukhin *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST Special Publication 800-22 Rev. 1a, Apr. 2010. doi: 10.6028/NIST.SP.800-22r1a.
[13] E. Kret and R. Schmidt, "The PQXDH key agreement protocol," *Signal*, 2023.
[14] M. Marlinspike and T. Perrin, "The Sesame algorithm: session management for asynchronous message encryption," *Revision*, vol. 2, 2017.