KaleidoTalk

Secure End-to-End Encrypted Chat with Cover Traffic

Version 3.0

Technical Whitepaper

Bangze Han

https://kaleidotalk.hanbangze.tech

2026-07-18

Repository: https://github.com/hbzsoft/KaleidoTalk

Abstract

KaleidoTalk is an open-source, end-to-end encrypted instant messaging application designed with privacy as its foundational principle. It employs a defence-in-depth cryptographic architecture combining Ed25519 identity keys, X25519 key exchange, AES-256-GCM symmetric encryption, and HKDF-SHA256 key derivation to ensure that only message participants can read message content. In addition to content encryption, KaleidoTalk implements *cover traffic* — a fixed-length packet protocol with random padding and randomized heartbeat intervals — to resist traffic analysis based on packet size and timing patterns. Version 3.0 introduces X25519 automatic key rotation with 24-hour expiry, an irreversible account freeze mechanism backed by Ed25519 recovery certificates, and a modern bubble-chat graphical user interface. This whitepaper presents the complete system architecture, cryptographic protocol, network design, security properties, and operational details of KaleidoTalk 3.0.

Contents

1 Introduction

1.1 Motivation

In an era where digital surveillance and data monetisation have become pervasive, private communication is no longer a default — it is an exception that must be explicitly engineered. Existing mainstream messaging platforms often employ end-to-end encryption only selectively, retain metadata for analytics, or rely on centralised certificate authorities that introduce trust single points of failure.

KaleidoTalk was built from scratch to address these shortcomings. It is guided by a manifesto[1] that asserts:

- End-to-end encryption is non-negotiable.
- Code must be open source for public audit.
- Users, not certificate authorities, control trust.
- Metadata (packet sizes and timing) must also be protected.
- The software must be free of ads, trackers, and data mining.

1.2 Design Goals

The design of KaleidoTalk is guided by the following objectives:

1. **Confidentiality**: Messages must be readable only by the intended recipient.
2. **Forward secrecy**: Compromise of a long-term key must not expose past messages.
3. **Authentication**: The recipient must be able to verify the sender's identity.
4. **Metadata protection**: Traffic analysis based on packet sizes and timing must be impeded.
5. **Self-sovereign trust**: Identity verification uses out-of-band fingerprint comparison, not centralised CAs.
6. **Key rotation**: Ephemeral keys are rotated automatically to limit the exposure window of key compromise.
7. **Account safety**: An irreversible freeze mechanism prevents unauthorised access even after credential compromise.

1.3 What is New in Version 3.0

- **X25519 key rotation**: Automatic 24-hour rotation cycle; the server stores multiple key versions for smooth transitions.
- **Irreversible account freeze**: An Ed25519 recovery certificate is generated during registration; if the account is compromised, the user can permanently freeze it.
- **Modern bubble-chat GUI**: CustomTkinter-based interface with contact list, avatars, unread badges, trust indicators, and dual-colour message bubbles.
- **Dual private-key storage modes**: Server-hosted (encrypted, for cross-device login) or local-only storage.
- **Server configuration file**: Configurable via `config.json` (host, port, security parameters, etc.).
- **Client configuration persistence**: Window size, theme, auto-connect preferences are saved locally.

[1] See `MANIFESTO.md` in the project repository.

2 System Architecture

2.1 High-Level Overview

KaleidoTalk follows a classical client-server topology with a custom, security-hardened protocol stack. The server acts as a message relay and key directory; it never has access to plaintext message content.

Presentation Layer `chat_gui.py` CustomTkinter bubble UI
Client Logic Layer `chat_client.py` — Connection, encryption, key mgmt
Cover Traffic Layer `padding.py` Fixed-length packets, random padding, fragmentation, heartbeat
Transport Layer TLS 1.2+ (self-signed RSA 2048)
Network Layer TCP/IP

Figure 1: Client Protocol Stack

Command Handling `server_commands.py`
Session & Security `server_session.py` HMAC auth, anti-replay, DoS
Cover Traffic Layer `padding.py` Fixed-length packets, heartbeat
Transport Layer TLS 1.2+ (self-signed RSA 2048)
Data Persistence `server_storage.py` JSON files, thread-safe

Figure 2: Server Protocol Stack

2.2 Module Layout

The source code is organised into three packages under `src/`:

`common/` Shared modules used by both client and server:

- `crypto_utils.py` All cryptographic primitives (key generation, serialisation, ECDH, AES-GCM, signing, BIP39 fingerprints).
- `padding.py` Cover traffic: fixed-length packet encapsulation, fragmentation/reassembly, heartbeat scheduling.
- `network.py` Thin length-prefixed JSON framing for internal server use.

`server/` Server-side components:

- `config.py` Configuration loader (`config.json`).
- `server.py` Main entrypoint: TLS setup, listener loop, client handler threads, `PaddedSocket` wrapper.
- `server_session.py` Session management, HMAC authentication, replay protection, DoS rate limiting, offline message queue.
- `server_commands.py` Command dispatch: registration, login, messaging, key rotation, account freeze, administration.

- `server_storage.py` Thread-safe JSON file persistence for users, keys, invites, bans.

`client/` Client-side components:

- `chat_client.py` Core client logic: TLS connection with TOFU trust, registration, login, message encryption/decryption, key rotation, trust database, heartbeat.
- `chat_gui.py` Graphical interface: contact list in left panel, bubble-chat in right panel, registration/login forms, trust verification dialogs.

3 Cryptographic Design

3.1 Key Material Hierarchy

Table 1: Cryptographic Components

Component	Algorithm	Key Size	Purpose
Identity key	Ed25519	256-bit	Signing sender identity
Key-exchange key	X25519	256-bit	ECDH shared-secret agr ment
Symmetric cipher	AES-256-GCM	256-bit	Message encryption and thentication
Key derivation	HKDF-SHA256	256-bit output	Derive AES key & nonce fr ECDH
Password hashing	PBKDF2-SHA256	256-bit output, 600k iterations	Server-side password verifi tion
Transport layer	TLS 1.2+, RSA 2048	2048-bit	Protect client-server link
Fingerprint encoding	BIP39 (2048-word list)	66-bit (6 words)	Human-readable identity v ification

3.2 Identity Keys (Ed25519)

Each client generates a long-term Ed25519 key pair at registration. The private key is used to sign messages (providing sender authentication and non-repudiation), and the public key is the user's canonical identity. Ed25519 was chosen over ECDSA for its resistance to side-channel attacks, small signatures (64 bytes), and deterministic signing that avoids nonce-reuse vulnerabilities.

Listing 1: Ed25519 key generation

```
1 private_key = ed25519.Ed25519PrivateKey.generate()
2 public_key  = private_key.public_key()
```

The fingerprint of a user's Ed25519 public key is displayed as six BIP39 English words for easy out-of-band verification (see Section 4).

3.3 Key-Exchange Keys (X25519)

Each client maintains one or more X25519 key pairs for ECDH-based shared-secret negotiation. In version 3.0, keys are automatically rotated every 24 hours. The server stores a list of active keys (with unique IDs and creation timestamps) so that messages encrypted with an older key can still be decrypted during the rotation transition window.

3.4 Message Encryption Protocol

The message encryption flow is as follows (see Figure 3):

1. The sender generates an *ephemeral* X25519 key pair. This key is used only for this single message, ensuring forward secrecy.
2. The sender performs ECDH using the ephemeral private key and the recipient's latest X25519 public key (fetched from the server), producing a shared secret S.
3. A random 16-byte salt is generated, and HKDF-SHA256 derives a 256-bit AES key K and a 96-bit nonce N from S and the salt (with info string `"kaleido-msg"`).
4. The plaintext is encrypted with AES-256-GCM using K and N, producing ciphertext C and an authentication tag T.
5. The sender signs the concatenation `eph_pub` $\|$ C $\|$ T with their Ed25519 private key, producing signature Σ.
6. The assembled packet (eph_pub, ciphertext, tag, nonce, salt, signature, optional key_id) is serialised as JSON and transmitted through the server.

$$
\begin{aligned}
S &= \text{ECDH}(\text{ephemeral\_priv}, \text{recipient\_x25519\_pub}) \\
K, N &= \text{HKDF-SHA256}(S, \text{salt}, \text{info} = \text{"kaleido-msg"}) \\
C, T &= \text{AES-256-GCM-Encrypt}(K, N, \text{plaintext}) \\
\Sigma &= \text{Ed25519-Sign}(\text{sender\_id\_priv}, \text{eph\_pub} \,\|\, C \,\|\, T)
\end{aligned}
$$

Figure 3: Message encryption process. Forward secrecy is guaranteed by the ephemeral X25519 key.

3.5 Decryption and Verification

1. The recipient receives the packet and parses the JSON fields.
2. The sender's Ed25519 public key (previously fetched from the server) verifies Σ over eph_pub $\|$ C $\|$ T. Rejection occurs if signature verification fails.
3. The recipient performs ECDH using *their own* X25519 private key (selected by `key_id` if present) and the sender's ephemeral public key, recovering the same shared secret S.
4. HKDF-SHA256 derives K and N from S using the received salt.
5. AES-256-GCM decrypts C using K, N, and T. Authentication failure indicates tampering.

The ephemeral public key is transmitted in the clear (it is not secret), but the shared secret S is known only to the sender and recipient because ECDH requires the private key of one side.

3.6 Forward Secrecy

Each message uses a fresh ephemeral X25519 key pair. The ephemeral private key is discarded immediately after encryption. Consequently, even if an attacker later compromises the long-term identity key or the recipient's X25519 private key, they cannot decrypt past messages because the ephemeral private key is no longer available. This property is known as *perfect forward secrecy* (PFS).

3.7 Key Rotation

In version 3.0, X25519 key pairs are automatically rotated every 24 hours. During rotation:

- The client generates a new X25519 key pair locally.
- The new public key is sent to the server with an Ed25519 signature for authentication.
- If server-hosted key storage is enabled, the new private key is encrypted with the user's password and stored on the server.
- The server inserts the new key at the head of the user's key list, preserving older keys for ongoing message delivery.

- The client includes a `key_id` in each encrypted message to indicate which of the recipient's keys was used.

This design ensures that a leaked X25519 private key only compromises messages in a narrow window (at most 24 hours), rather than all past and future communications.

3.8 Server Public-Key Encryption

The server itself possesses an X25519 key pair. When a client sends sensitive data (e.g., password during registration/login), it is encrypted using the server's X25519 public key with the same ECDH + HKDF + AES-256-GCM scheme. This prevents plaintext passwords from appearing on the wire even within the TLS tunnel, providing defence-in-depth against TLS termination attacks.

3.9 Password Storage

Server-stored passwords use PBKDF2-HMAC-SHA256 with 600,000 iterations and a random 16-byte salt. The storage format is:

`PBKDF2$<hex(salt)>$<hex(derived_key)>`

This high iteration count makes brute-force attacks computationally expensive. A weak-password blacklist (20 common passwords) is also enforced at registration.

4 Trust Model

4.1 Trust On First Use (TOFU)

KaleidoTalk uses a TOFU trust model for both TLS server certificates and user public keys:

- On the *first connection* to a server, the client displays the TLS certificate fingerprint (SHA-256 hash) encoded as six BIP39 English words. The user is prompted to verify this fingerprint through an independent secure channel (e.g., in person or via a phone call). Once confirmed, the fingerprint is stored in a local trust database.
- On *subsequent connections*, the client automatically compares the received fingerprint against the stored value. A mismatch triggers an alert and disconnection.

This eliminates reliance on external certificate authorities while providing strong authentication after the initial verification.

4.2 BIP39 Fingerprint Encoding

Rather than displaying raw hex strings, KaleidoTalk encodes the SHA-256 fingerprint of a public key into human-readable English words using the BIP39 standard word list (2048 words). The encoding divides the fingerprint bytes into 11-bit chunks, each mapping to a word index:

$$\text{words} = \{\text{wordlist}[\text{bits}_{i:i+11}] \mid i = 0, 11, 22, \dots\}$$

With six words (66 bits), the fingerprint is easily compared by humans while providing sufficient entropy to prevent collision attacks in practice.

4.3 User Identity Verification

When chatting with a contact for the first time, KaleidoTalk displays that contact's Ed25519 public key fingerprint as six BIP39 words. Users are encouraged to verify these words through a secure channel. Once verified, the trust relationship is stored in the local trust database, and subsequent messages are decrypted automatically without re-verification.

4.4 Trust Database Integrity

The local trust database (stored in `local_keys/trusted_fingerprints.json`) is protected by an HMAC-SHA256 key generated at client startup and stored in `local_keys/.hmac_key`. Before loading the trust database, the client recomputes the HMAC and compares it against the stored signature. If the integrity check fails, the database is considered tampered with and all trust relationships are reset.

5 Cover Traffic Protocol

5.1 Motivation

Many encrypted messaging applications protect message content but leak metadata through observable packet characteristics. An eavesdropper who cannot read ciphertext can still infer behavioural patterns from:

- **Packet sizes**: Variable-length messages reveal information about message length and content type.
- **Packet timing**: The frequency and pattern of messages reveal active periods, response times, and potentially even social graph structure.

KaleidoTalk's cover traffic layer addresses both channels.

5.2 Fixed-Length Packet Format

All data transmitted between client and server is encapsulated in fixed-length packets of exactly **2048 bytes**. Each packet has the following structure:

Table 2: Fixed-length packet format (2048 bytes total)

Field	Offset	Size	Description
Type	0	1 byte	Packet type identifier
Length	1–2	2 bytes	Real data length (big-endian)
Seq	3–4	2 bytes	Fragment sequence number
Total	5–6	2 bytes	Total fragment count
Payload	7–N	Variable	Real data (up to 2041 bytes)
Padding	N–2047	Remaining	Random bytes

5.3 Packet Types

`0x00` **Padding** Pure cover traffic; no real data attached.
`0x01` **Data** Complete message fitting within one packet.
`0x02` **Fragment first** First fragment of a multi-packet message.
`0x03` **Fragment mid** Intermediate fragment.
`0x04` **Fragment last** Final fragment.

5.4 Fragmentation and Reassembly

Messages larger than 2041 bytes (2048 minus 7-byte header) are transparently fragmented by the sender and reassembled by the receiver using the ‘FragmentReassembler‘ class. Each fragment carries its sequence number and total count, enabling unordered delivery and lost-fragment detection. A 30-second timeout discards stale fragments to prevent memory exhaustion attacks.

5.5 Heartbeat Traffic

Both client and server generate periodic padding packets at randomised intervals to mask the absence of real traffic, a technique known as *traffic morphing*:

$$\text{interval} = 5.0 + \text{Uniform}(-1.67, +1.67) \text{ seconds}$$

The base interval is 5 seconds with a jitter of $\pm\frac{1}{3}$ of the base interval. This constant stream of fixed-size packets makes it impossible for a passive observer to distinguish between active conversation and idle standby periods based on traffic volume alone.

Listing 2: Heartbeat scheduling

```
def next_interval():
    jitter = BASE_INTERVAL * JITTER_RATIO
    return BASE_INTERVAL + random.uniform(-jitter, jitter)
```

5.6 Padding Mechanism

All data packets use random bytes (generated by `os.urandom`) for padding rather than zero-padding, preventing an observer from distinguishing padding from real data by entropy analysis. Pure padding packets contain only the header followed entirely by random bytes.

5.7 Server-Side Processing

On the server side, a ‘PaddedSocket‘ wrapper transparently handles the conversion between the fixed-length packet protocol and the internal JSON framing used by command handlers. Each client connection has a dedicated heartbeat thread and a single ‘PaddedSocket‘ instance, which also serves as the key for session tracking in the server’s global state dictionaries.

6 Session and Authentication Protocol

6.1 Login Flow

The login process proceeds through the following stages:

1. **TLS handshake**: The client establishes a TLS 1.2+ connection and performs TOFU certificate verification.
2. **Key retrieval**: The client requests the server’s public keys (`get_server_pubkey`) and registration policy (`get_reg_policy`).
3. **Password transmission**: If server-hosted key storage is used, the password is encrypted with the server’s X25519 public key and sent to the server. For local-only storage, the server issues a signature challenge.
4. **Session creation**: Upon successful authentication, the server creates a session with a fresh random session ID and a 32-byte random session key.
5. **Session key delivery**: The session key is encrypted with the client’s X25519 public key (using ECDH + HKDF + AES-256-GCM) and sent to the client.
6. **Private key retrieval**: If server-hosted storage is enabled, the user’s encrypted private keys are returned and decrypted locally with the password.

6.2 Authenticated Requests (HMAC)

After login, all privileged commands are authenticated using HMAC-SHA256:

$$\text{HMAC} = \text{HMAC-SHA256}(\text{session\_key}, \text{session\_id} \parallel \text{seq} \parallel \text{timestamp} \parallel \text{canonical\_json}(\text{data}))$$

Each request includes:

- `session_id`: Identifies the session.
- `seq`: Monotonically increasing sequence number (per-session).
- `timestamp`: Client time in milliseconds, adjusted for server-client clock skew.

6.3 Replay Protection

The server maintains, for each session:

- The last-seen sequence number `last_seq`. Any request with a sequence number $\leq$ `last_seq` is rejected.
- A timestamp cache. Timestamps within the 5-minute time window are tracked; replayed timestamps are rejected.
- A 5-minute maximum clock skew tolerance. Requests outside this window are discarded.

This dual protection (sequence number + timestamp) prevents both trivial replay and delayed replay attacks.

6.4 DoS Protection

The server implements per-IP rate limiting for registration and login attempts (configurable; defaults to 10 registration attempts and 20 login attempts per 60-second window). Exceeding the limit results in an automatic IP ban lasting 1 hour (configurable). The IP ban list is persisted to disk and reloaded on server restart.

6.5 Session Lifecycle

- Sessions exist only for the duration of the TCP/TLS connection. No session persistence mechanism is provided (messages are encrypted end-to-end and the server does not need to re-authenticate after disconnection).
- Only one active session per user is permitted. If a user logs in from a second device, the existing session is forcibly terminated with a `force_logout` message.
- When a user disconnects, the session is immediately cleaned up from server memory.

7 Irreversible Account Freeze Mechanism

7.1 Motivation

If a user's password is forgotten or their credentials are stolen, they need a way to prevent an attacker from accessing their account permanently. Traditional "password reset" mechanisms rely on email or SMS, which introduces trust dependencies and potential for social engineering.

7.2 Recovery Certificate

During registration, an Ed25519 key pair (the *recovery key*) is generated and saved locally as `local_keys/<username>_recovery.priv`. The public half of this key is stored on the server as part of the user's key data. This key is entirely independent of the user's password and identity key.

7.3 Freeze Protocol

To freeze an account, the user (or anyone in possession of the recovery key) sends:

$$\text{message} = \text{target\_username} \parallel \text{timestamp} \parallel \text{nonce}$$

$$\Sigma = \text{Ed25519-Sign}(\text{recovery\_priv}, \text{message})$$

The server verifies:

1. The target user exists and has a recovery public key on record.
2. The timestamp is within the 5-minute window.
3. The nonce has not been used before (anti-replay, with a rolling cache of 100 nonces).
4. The Ed25519 signature is valid against the stored recovery public key.

Upon successful verification, the server sets the `frozen` flag on the account to `True`, which:

- Prevents any future login attempts (`login` commands return `"Account frozen"`).
- Prevents message delivery to or from the frozen account.
- Forcefully terminates any currently active sessions.
- Is **cryptographically irreversible**: no server command or database edit can unset the `frozen` flag without invalidating the verifiability assumption[2].

7.4 Standalone Freeze Tool

A standalone tool (`freeze_account.py`) is provided that connects to the server directly (without the full GUI client) and issues the freeze command. This allows users to freeze their accounts even if they no longer have access to the full client software.

8 Network Protocol Specification

8.1 Command Set

All client-server communication is JSON-based, transported inside the fixed-length packet layer. The following commands are defined:

[2]The frozen flag is a regular database field, but the ethical design guarantees that re-enabling a frozen account would require manual database editing, which can be audited through code review. The protocol itself provides no "unfreeze" command.

Table 3: Server Commands

Command	Auth	Description
`get_server_pubkey`	No	Retrieve server's Ed25519 and X25519 public keys
`get_reg_policy`	No	Check if invite-code registration is required
`reg_user`	No	Register new account with keys and optional invite code
`login`	No	Authenticate and establish session
`challenge_response`	No	Respond to signature challenge (local-key mode)
`message`	Yes	Send encrypted message to another user
`list_users`	Yes	List currently online users
`get_pubkey`	Yes	Get another user's public keys
`get_my_keys`	Yes	Retrieve own key list and encrypted private keys
`rotate_key`	Yes	Submit a new X25519 public key for rotation
`freeze_account`	Yes (recovery key)	Permanently freeze an account
`logout`	Yes	Terminate current session
`admin_ip`	No (loopback)	IP ban/unban/list (local administration)

8.2 Message Format

All commands use a uniform JSON request envelope:

Listing 3: Request envelope

```
{
    "cmd": "message",
    "session_id": "<hex session token>",
    "seq": 42,
    "timestamp": 1712345678000,
    "hmac": "<hex HMAC signature>",
    "data": { ... }
}
```

Responses follow a standard format:

Listing 4: Response envelope

```
{
    "status": "ok" | "error" | "challenge",
    "cmd": "message",
    "data": { ... },
    "error": "<error message if status is error>"
}
```

8.3 Message Relay

When a sender sends a message to a recipient:

1. The sender constructs the end-to-end encrypted packet (see Section 3) and sends it to the server with the recipient's username.
2. The server checks the recipient's freeze status and looks up the recipient's current socket.
3. If the recipient is online, the server relays the message immediately and returns `delivered` to the sender.
4. If the recipient is offline, the server stores the message in the pending queue and returns `offline_saved`.
5. Upon the recipient's next login, all queued offline messages are delivered before any other communication.

The server never decrypts or inspects the message payload; it is opaque binary data from the server's perspective.

9 Implementation Details

9.1 Technology Stack

- **Language**: Python 3.8+
- **Cryptography**: `cryptography` library (Rust-backed, with OpenSSL bindings)
- **GUI**: CustomTkinter (modern theme on top of Tkinter)
- **System Tray**: `pystray`
- **Image handling**: Pillow (for avatars)
- **Concurrency**: `threading` one thread per client on the server; separate network and GUI threads on the client
- **Persistence**: Plain JSON files with atomic write (*write-to-temp-then-rename*)

9.2 Concurrency Model

Server: The main thread accepts incoming TCP connections. Each accepted connection is handed off to a dedicated daemon thread. The listener runs in a tight loop with a `KeyboardInterrupt` guard. All shared state (client registry, session table, online users, IP attempt counters) is protected by per-data-structure `threading.Lock` instances.

Client: Three threads are active during operation:

1. **Main/GUI thread**: Handles UI events in the Tkinter event loop.
2. **Receiver thread**: Continuously reads fixed-length packets from the socket and dispatches messages to the GUI via callback.
3. **Heartbeat thread**: Periodically sends padding packets for cover traffic.

All network writes are serialised through a per-connection `send_lock` to avoid interleaving.

9.3 Data Persistence

All server data is stored as JSON files in the server working directory. The storage layer (`server_storage.py`) provides:

- Thread-safe reads and writes via per-file locks.
- Atomic writes using the `write-to-temp-then-os.replace` pattern, preventing partial writes from corrupting data.
- Automatic migration from legacy key format (single `x25519_pub`) to the new multi-key format (`x25519_keys` list), performed transparently on first load.
- Backfilling of `frozen` and `used_nonces` fields for pre-existing users.

The client stores:

- **Trust database**: `local_keys/trusted_fingerprints.json` + HMAC integrity file.
- **Client configuration**: `local_keys/client_config.json` (server address, window size, theme, auto-connect).
- **Recovery key**: `local_keys/<username>_recovery.priv` (Ed25519 PEM file).
- **Private keys (local mode)**: `local_keys/<username>_ed25519.priv` and `<username>_x25519.priv`.

9.4 Server Configuration

The server configuration is stored in `config.json` at the project root:

Listing 5: Default server configuration

```
{
    "host": "0.0.0.0",
    "port": 5555,
    "interactive": true,
    "log_level": "INFO",
    "max_packet_size": 2048,
    "security": {
        "ip_ban_duration": 3600,
        "register_limit": 10,
        "login_limit": 20,
        "time_window": 60,
        "session_time_window": 300,
        "max_message_size": 10485760
    }
}
```

10 Security Analysis

10.1 Threat Model

KaleidoTalk assumes the following adversary capabilities:

- **Network-level adversary**: Can observe, intercept, modify, replay, and drop network packets. Can perform traffic analysis.
- **Server compromise**: The server operator is assumed to be semi-honest (honest-but-curious). The adversary may have full access to server hard drives, databases, and memory. However, the adversary cannot modify the client software.
- **Endpoint compromise**: The adversary may compromise a user's device (but not both endpoints simultaneously).
- **Man-in-the-middle**: The adversary may attempt to impersonate the server or another user.

KaleidoTalk does *not* protect against:

- Traffic correlation attacks at the IP level (the timing of the entire encrypted packet stream can still reveal communication patterns between specific IP addresses).
- Compromise of both communicating devices simultaneously.
- Physical coercion or rubber-hose cryptanalysis.

10.2 Security Properties

1. **End-to-end confidentiality**: Messages are encrypted with AES-256-GCM before transmission. The server never possesses the decryption key. An adversary with full server access cannot read message content.

2. **Perfect forward secrecy**: Each message uses an ephemeral X25519 key pair. Compromise of long-term keys does not expose past messages.
3. **Sender authentication**: Every encrypted message is signed with the sender's Ed25519 identity key. The recipient can verify the signature using the sender's public key.
4. **Message integrity**: AES-256-GCM provides authenticated encryption. Any tampering with ciphertext results in authentication failure during decryption.
5. **Replay protection**: Dual-layer protection (sequence numbers and timestamp caching) prevents both immediate and delayed message replays.
6. **Denial-of-service resilience**: Per-IP rate limiting with automatic banning mitigates registration and login flooding attacks.
7. **Metadata protection**: Fixed-length packets and heartbeat traffic prevent traffic analysis based on packet size and timing.
8. **Trust agility**: TOFU trust model eliminates dependence on centralised certificate authorities.

10.3 Potential Weaknesses

1. **First-use weakness**: The TOFU model is vulnerable to a man-in-the-middle attack during the *first* connection if the user negligently accepts an incorrect fingerprint. The BIP39 word display mitigates this by making fingerprints human-verifiable.
2. **No deniability**: Ed25519 signatures provide non-repudiation, which means a message can be cryptographically proven to have been sent by a specific user. This is intentional for accountability but may be undesirable in some privacy contexts.
3. **Metadata residual risk**: While cover traffic obscures packet timing and size, an adversary who can observe the entire network can still see that two IP addresses are communicating with a known KaleidoTalk server at specific times.
4. **No forward secrecy for session keys**: The session key (used for HMAC authentication) is derived once per login and does not rotate during the session. Compromise of the session key during an active session would allow request forgery.
5. **JSON-based storage**: The server uses JSON files for persistence. While atomic writes prevent corruption, the design does not scale to thousands of concurrent users and is not intended for production deployment at scale.

11 Performance Characteristics

11.1 Overhead Analysis

The cover traffic protocol introduces the following overhead:

- **Packet overhead**: 7 bytes of header + 0–2041 bytes of padding per 2048-byte packet.
- **Minimum bandwidth (idle)**: One 2048-byte packet every 3.3–6.7 seconds $\approx$ 310–620 bytes/s.
- **Maximum throughput**: $\frac{2041 \text{ bytes}}{2048 \text{ bytes}}\times$ network bandwidth = 99.66% efficiency for large messages.
- **Cryptographic overhead**: Each AES-256-GCM message adds a 12-byte nonce, 16-byte authentication tag, and the sender's ephemeral X25519 public key (32 bytes) approximately 72 bytes of overhead per message (excluding signature and base64 encoding expansion).

11.2 Scalability Considerations

The current implementation uses a thread-per-client model. For a small-scale private deployment (tens of concurrent users), this is adequate. For larger deployments, an asynchronous I/O

model (e.g., `asyncio`) would be more appropriate. The protocol and cryptographic design are independent of the concurrency model and would not require modification.

12 Comparison with Existing Solutions

Table 4: Feature comparison with other secure messaging systems

Feature	Signal	Tox	Matrix	KaleidoTalk
End-to-end encryption	Yes	Yes	Optional	Yes
Forward secrecy	Yes (ratchet)	Yes (ratchet)	Partial	Yes (ephemeral)
Key rotation	Yes	No	No	Yes (24h auto)
Cover traffic	No	No	No	Yes
Open source	Yes	Yes	Yes	Yes (GPL v3)
No CA dependency	No	Yes	No	Yes (TOFU)
Account freeze	No	No	No	Yes (irreversible)
Centralised server	Yes	No	Yes	Yes
Offline messages	Yes	No	Yes	Yes
BIP39 fingerprint	Partial	Partial	No	Yes

KaleidoTalk's distinguishing features are its cover traffic protocol for metadata protection and its irreversible account freeze mechanism, both of which are absent from mainstream alternatives.

A Appendix A: Protocol Frame Examples

A.1 Heartbeat / Padding Packet

Listing 6: A pure padding packet (hex dump of first 64 bytes)

```
00 00 00 00 00 00 00  [header: type=0x00, len=0, seq=0, total=0]
<random bytes filling to 2048 bytes>
```

A.2 Data Packet (Complete)

Listing 7: A data packet header (hex)

```
01 00 21 00 00 00 01  [header: type=0x01, len=33, seq=0, total=1]
{"cmd": "get_server_pubkey"}<random padding to 2048 bytes>
```

Encrypted Message Payload (JSON, base64-encoded fields)

Listing 8: Encrypted message sent over the wire

```
{
    "type": "msg",
    "sender": "alice",
    "payload": "{\"eph_pub\":\"A...\",\"ct\":\"B...\",\"tag\":\"C
       ...\",\"nonce\":\"D...\",\"sig\":\"E...\",\"salt\":\"F...\",\"
       key_id\":\"alice_1712345678\"}",
    "timestamp": 1712345678
}
```

B Appendix B: Automated Testing

The `test/` directory contains integration and smoke tests:

- `test_key_rotation_smoke.py`: Verifies local X25519 key generation, encryption, decryption, and rotation.
- `test_e2e_rotation.py`: End-to-end test simulating registration, login, message exchange, and key rotation with a running server.
- `test_freeze_smoke.py`: Tests recovery key generation, signature creation, and freeze request construction.
- `test_freeze_e2e.py`: End-to-end test of the account freeze protocol including nonce replay detection.

All tests connect to a local server instance and validate the protocol at the network level.

C Appendix C: Deployment Guide

1. Install dependencies: `pip install -r requirements.txt`
2. Start the server: `python run_server.py`
   - On first start, provide an admin password (used to encrypt server private keys).
   - TLS certificate and keys are auto-generated in `server_keys/`.
   - Server configuration can be customized in `config.json`.
3. Start the client: `python run_client.py`
   - Connect to the server address (default `127.0.0.1:5555`).
   - Verify the TLS certificate fingerprint (displayed as six BIP39 words).
   - Register an account and save the recovery key.
   - Start chatting.
4. Administration: `python admin.py` for invite codes, bans, and user management.
5. Emergency account freeze: `python freeze_account.py -server <addr> -username <user> -recovery-key <path>`

References

[1] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[2] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security," RFC 7748, Internet Engineering Task Force, Jan. 2016.

[3] D. A. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," NIST, 2004.

[4] H. Krawczyk, "Cryptographic Extraction and Key Derivation: The HKDF Scheme," in *Advances in Cryptology — CRYPTO 2010*, pp. 631–648, Springer, 2010.

[5] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe, "BIP-0039: Mnemonic Code for Generating Deterministic Keys," Bitcoin Improvement Proposal, 2013.

[6] T. Perrin and M. Marlinspike, "The Double Ratchet Algorithm," Signal Technical Report, Nov. 2016.

[7] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix Anonymity System," in *Proceedings of the 26th USENIX Security Symposium*, pp. 1199–1216, Vancouver, 2017.

[8] A. Back, "Hashcash — A Denial of Service Counter-Measure," Technical Report, 2002.