Secure Chats — Privacy & Security White Paper

One page. Two ideas. Every message end-to-end encrypted.

MARCEL CRASMARU — FOUNDER, SECURE CHATS

ABSTRACT

Secure Chats reduces the trust the user must place in the operator to two auditable claims. **The server is small enough to specify as an LLM prompt**, so anyone can regenerate a compatible server and check that the reference one behaves the same way. **Key exchange can happen entirely between two phones via QR code**, so the shared secret never crosses the internet in either its plaintext or its encrypted form. Together they shrink the attack surface to the messaging clients themselves, which are open source.

1. Threat model

Adversary	Defended?	How
Passive network observer (ISP, coffee-shop Wi-Fi)	Yes	All traffic is TLS + encrypted payloads. Wire carries opaque blobs.
Malicious or compromised server operator	Yes, for message content	Server holds no plaintext, no keys, no display names, no cleartext tokens. Alternative servers are one prompt away.
Man-in-the-middle during key exchange	Yes — prevented on the QR flow, detected on the server flow	**QR flow:** public shares travel between two cameras, not over the internet — there is no channel for an attacker to sit in. **Server flow:** comparing the 24-hex-digit fingerprint (*Verify End-to-End Encryption*) out of band detects any key substitution after the fact. The defence only holds if the users actually compare the codes; silence is not safety.
Endpoint compromise (someone	No	Out of scope. Any messenger is defeated by an owned device. Use OS-level device encryption and disappearing messages to limit blast radius.

Adversary	Defended?	How
unlocks your phone)		
Traffic-pattern analysis (who talks to whom, when)	Partial	The server sees connection metadata by definition. Rate-limit and TLS mitigate; onion routing would be needed to fully address.

2. Server: small enough to be a prompt

Most messengers ask you to trust the operator's server the way you trust a bank. Secure Chats inverts that: the server does so little that its entire behaviour fits in a self-contained LLM prompt. Blog entry 5 walks through the exercise of asking Claude to write the prompt, then handing that prompt to Gemini to one-shot a runnable server. Because the prompt is the contract, anyone can:
Regenerate a compatible server from scratch in Python or Node.js (Secchats-tech/VibeServer).
Diff its behaviour against the published JSON API.
Run it locally and point the mobile clients at it — the same messages will round-trip.
What the reference server is required to store, and what it is required *not* to store, are baked into the prompt's *hard invariants* and *definition of done*:

```
-- What the server keeps
principals(id, kind, auth_token_hash, created_at)
notifications(id, to_principal, kind, from_user, group_id,
              protocol, algorithm, crypto_data, enc_content, ts_ms)

-- Invariants the prompt bakes in
* authToken is never persisted; only HMAC-SHA256(secret, token) is
  stored, compared in constant time.
* Display names live only inside enc_content. The server never
  indexes, logs, or echoes them; the name column is "" or null.
* crypto_data / enc_content are opaque blobs on the wire and on
  disk; the server has no keys with which to open them.
* No plaintext tokens or message bodies in any log line.
```

Because the invariants are part of the prompt, they are part of the **test suite** every implementation must pass. The end-to-end proof is a single command: `sqlite3 db.sqlite '.dump'` after a register → send round-trip must show no plaintext token, no display name, and no readable message content.

Reference implementations

VibeServer — the two prompts (Python & Node.js) | ChatJavaUI — the Java reference client

3. Client: face-to-face key exchange, no server involvement

Server-brokered key exchange has a well-known weakness: whoever runs the relay could substitute keys and read every subsequent message. Secure Chats sidesteps it with a **QR-code Diffie-Hellman handshake carried over any channel that can move images back and forth between two people who trust each other's face and voice** — the two phones do not need to be on the same network, and Secure Chats does not have to be the app doing the transport.

The clearest case is two phones sitting on the same table: Alice's phone displays her QR, Bob's camera reads it, Bob's phone displays the reply, Alice's camera reads it back. Nothing about the handshake ever hits the internet, and the man-in-the-middle attack cannot happen.

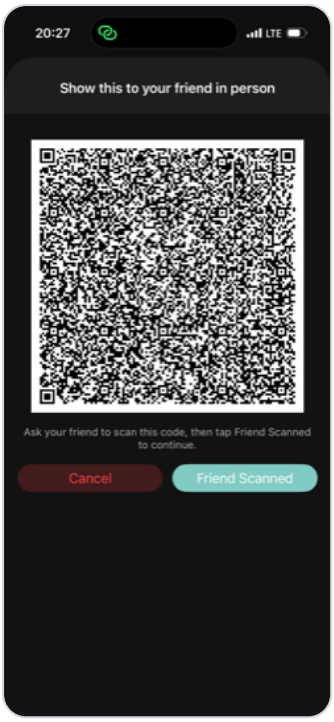
The same handshake also works over any live, image-carrying channel between two parties who can recognise each other:

Video call (WhatsApp, Signal, Zoom, FaceTime, Google Meet, Jitsi — any of them). Alice holds her QR up to her phone's front camera; Bob scans it from his screen; then they reverse. Because the parties see and hear each other in real time, a swap of the QR in transit would be visible — the verification code in step 6 would still fail to match if any middlebox managed to re-encode the image.

Screen share. Alice screen-shares the QR view over a call; Bob scans it directly off his own screen.

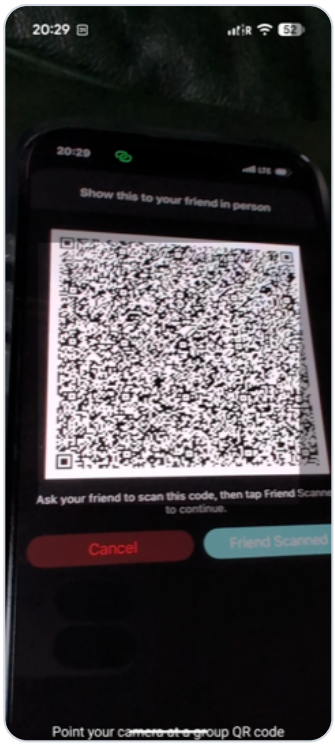
Printed or photographed QRs. A screenshot mailed over an already-trusted channel (e.g. an existing encrypted chat, a company Signal group, a printed piece of paper) works too. The integrity guarantee still rests on the verification-code compare at the end.

In every case, the shared secret is derived *only* on the two phones. Both phones then display the first 96 bits of SHA-256(shared secret) as a 24-hex-digit code — if the codes match, both sides derived the same key and no attacker was able to substitute their own material during transport. If they don't match, the pairing is aborted and no channel is created.



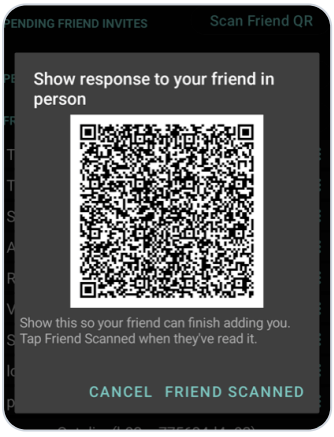
STEP 1 — ALICE

Taps *Friend Invite QR*. Her phone generates a fresh DH key pair and shows the public half as a QR.



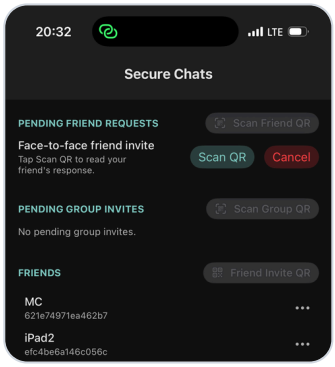
STEP 2 — BOB

Taps *Scan Friend QR* and reads Alice's public share through the camera.



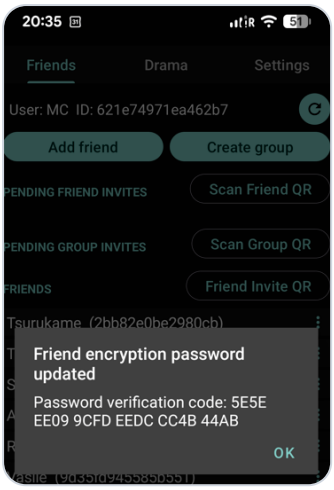
STEP 3 — BOB

Generates his own DH pair, derives the shared secret locally, and shows his public half as a response QR.



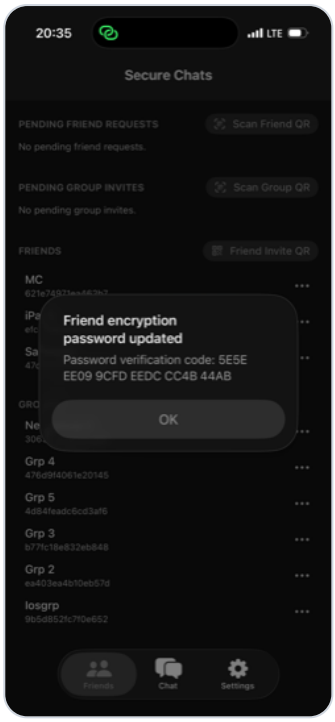
STEP 4 — ALICE

Scans Bob's reply. Her phone finishes the handshake and derives the same shared secret Bob just did.



STEP 5 — BOB

His phone shows a 24-hex-digit code — the first 96 bits of SHA-256 over the shared key.



STEP 6 — ALICE

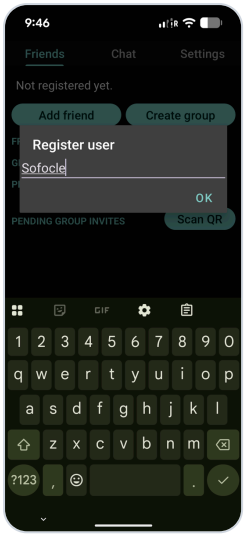
Her phone shows the same code. Matching codes prove both phones derived the same password. Tap OK; the chat is ready.

Rescanning a fresh pair of QRs later — in person or over any of the image-carrying channels above — **rotates the key without deleting the chat**. If a device is lost or you simply want fresh keys, meet again (or hop on a video call) and the message history stays put while encryption starts over. Auth tokens are stripped from the QR payload before encoding, so a bystander with a long-lens camera or a stray screen recording cannot hijack an account.

4. Normal flows (over the server)

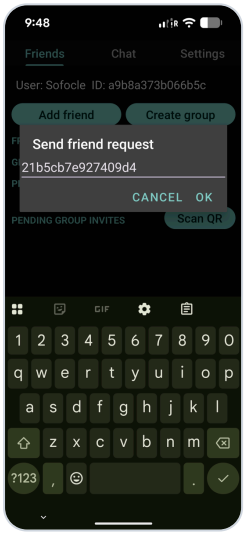
Face-to-face is the paranoid path. The everyday path uses the server as a queue for encrypted material and nothing more.

Registration

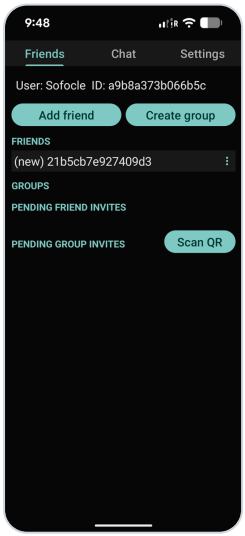


`POST /register` returns a random hex `userId` and an `authToken`. The nickname is chosen locally and never sent to the server as a standalone field — it travels later, embedded inside encrypted messages.

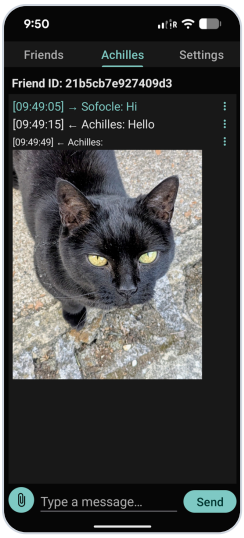
Adding a friend



Enter your friend's user id. Your app calls `POST /connect` with your half of an ECDH exchange — only the *public* value crosses the wire.

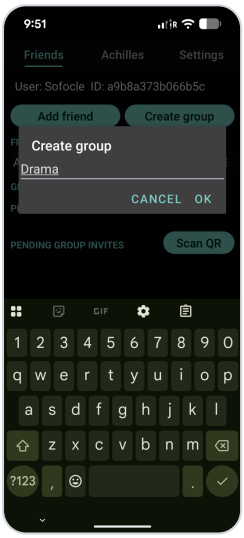


Your friend accepts; their app replies with `POST /connect2` carrying their public value.

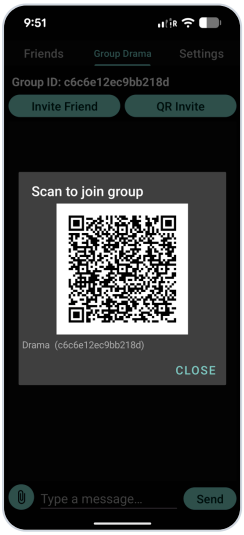


Each device derives the same shared secret locally. Every subsequent message on `POST /postmsg` is AES-256 encrypted with it.

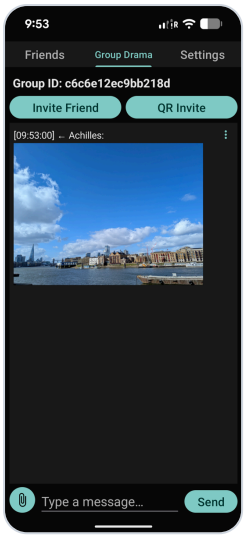
Group chat



`POST /register2` creates the group and returns a random `groupId`. The group key is generated locally.



The group key can be shared over the server via `POST /groupinv` (wrapped in each recipient's friend key) or in person via QR — server-free.



`POST /postgroup` and `POST /pollgroup` move ciphertext for every member. The server never holds the group key.

5. How to verify the claims yourself

Server-side. Point either VibeServer build (Python or Node) at a local SQLite file. Register a user, send a message, and run `sqlite3 db.sqlite '.dump'`. You will not find your nickname, your token, or your message.
Wire-side. Watch the traffic between the client and `secchats.com` with any TLS-terminating proxy. Every non-metadata field is a base64 blob. The published API tells you which fields are those blobs.
Client-side. The mobile apps are open source: SecureChats-Android and SecureChats-ios. Fingerprint the key from either app (*Verify End-to-End Encryption* in the friend menu) and compare to your peer's fingerprint. A mismatch means someone is between you.

6. Open source

All four components are published on GitHub:

ChatJavaUI · VibeServer · SecureChats-Android · SecureChats-ios

Because the server is a prompt and the clients are source-visible, no trust in the operator is required beyond running the transport and queuing ciphertext. Every claim in this paper is falsifiable with a terminal and a phone.

Companion documents: the Guide walks through the face-to-face flow with screenshots, the API spec is the contract every server implements, and blog entry 5 covers the one-shot LLM server experiment.