Project Deliverables

-z5522906

Github Repo -
https://github.com/jatinSharma-create/6441---Project---E2E-Chat

1. Aim

In this project I built a small end-to-end encrypted chat system to *show* the core security principles—confidentiality, integrity, and a (limited but still demo-able) availablity—plus a manual authentication step, instead of only talking theory. The design uses X25519 for key agreement, HKDF to derive session keys, and NaCl's XChaCha20-Poly1305 for authenticated encryption. I also plugged a kind of Double-Ratchet style layer into a Python TUI client and a minimal asyncio TCP relay. During testing I captured PCAP traces, deliberately triggered MAC verification fails, and measured how the relay behaves when one peer suddenly drops out. This report summarises the motivation and method, the evidence collected, issues and ethics I ran into, and what I actually learnt over ~30 hours (approx) of work.

2. Motivation & Need

The main motivation of this project was basically two-fold:

(1) I wanted to actually see and understand the moving pieces inside "secure chat", not just repeat the buzzwords; and

(2) I needed to demonstrate those parts clearly for the course criteria. Rather than trying to ship a full production-grade system (which would've been overkill and there was time constraint as well),

I targeted a minimal proof-of-concept that lines up directly with CIA plus authentication. So I kept the server - relay super simple and spent more time thinking through the crypto logic/reasoning. That was the whole goal, honestly.

3. Background, Crypto Choices & Quick Definitions

For the crypto stack I looked at a few modern primitives and picked ones that are both well-regarded and (kinda) straightforward to derive from Python. Briefly:

- **X25519 (Diffie–Hellman on Curve25519):** Fast scalar mult to derive a shared secret without exposing private keys. It's pretty much the de-facto standard in Signal/Noise etc, and PyNaCl gives a clean API, so I chose it over older stuff like RSA.
- **HKDF (HMAC-based Key Derivation Function):** takes that single shared secret and stretches it into lots of separate keys (root, chain, message…). The Double-Ratchet burns through keys a lot, so HKDF was necessary; otherwise I'd be reusing entropy which was bad news.
- **XChaCha20-Poly1305 (via NaCl `SecretBox`):** an AEAD cipher: ChaCha20 handles confidentiality, Poly1305 adds integrity (a MAC). The huge nonce space means I'm less likely to screw up nonce reuse, and the API is honestly hard to misuse (though not impossible).
- **Double-Ratchet (Signal-style):** after the first X25519 exchange, every message "ratchets" forward (basically another DH + HKDF hop) so we get forward secrecy and avoid key reuse. I only did a trimmed version—no skipped-message queue, minimal state—but it still rotates per-message keys.

I briefly thought about just using TLS on the socket or even using RSA, but that hides the exact pieces I wanted to learn about. MLS looked good too, but it's way too heavy for ~30 hours and a 2-person chat demo.

4. Problem Statement & Objectives

The core problem was to build a tiny chat system where only the two endpoints can actually read the messages, any tampering gets caught, and the middle relay can't be trusted at all. On top of that, I had to *prove* each security property with real evidence (PCAP dumps, logs, screenshots), not just hand-wavy claims.

Objectives:

- **Confidentiality:** No plaintext should appear on the network trace. I showed this in Wireshark PCAPs.
- **Integrity:** If I flip bits in a ciphertext, the MAC check must fail and the client refuses it.
- **Availability:** The relay process keeps running even when clients join/leave. It might drop a few packets, but it doesn't just die.
- **Authentication:** Users manually verify each other's public‑key fingerprints.
- **Forward Secrecy:** Implemented via the mini Double‑Ratchet. I didn't finish a proper replay/compromise demo though—I explained why later and what was missing.

(There's probably still a couple rough edges, but that was the target set of goals I aimed for.)

5. <u>Design Overview (high level)</u>

There are basically three moving parts:

1. **Client (Alice/Bob):** A Python TUI built with Rich. Each client makes a static X25519 keypair, prints its own + the peer's fingerprint, and then uses a kind of Double‑Ratchet to encrypt every outgoing message.
2. **Relay (server.py):** An untrusted asyncio TCP forwarder. It just keeps a tiny buffer of pubkeys and pipes framed messages around. No database, no disk writes, nothing complicated.
3. **Attacker (conceptual):** Can sniff packets, drop/replay them, or even flip bits. But they shouldn't be able to decrypt or forge legit ciphertexts (unless I seriously messed up).

The data‑flow diagram (threat_model.png) shows the trusted clients vs the untrusted relay/network, plus where an attacker would sit. Public keys + ciphertext go through the relay, but plaintext is meant to never leave the client processes. (If it does… that would be a bug.)

At the moment the model only spins‑up a single 1‑to‑1 Double‑Ratchet session between Alice and Bob. The relay server itself can already broadcast to heaps of sockets, but my client keeps just one active ratchet state, so talking to exactly one peer. If I want to scale to *N* users I'd pretty much need to hold N separate ratchet objects (one per peer) **or** swap in a proper group protocol like MLS, which is very heavier and I haven't looked into it yet.

6. Implementation Summary

crypto.py

- generate_keypair(), dh() (X25519), and a couple HKDF helpers (hkdf_root, hkdf_chain) to stretch secrets.
- RatchetSession: a trimmed Double-Ratchet—each message does a DH hop + HKDF to spit out fresh message keys. It's minimal (no skipped-msg queue etc.) but works for my aims.

server.py

- Uses asyncio.start_server; keeps an in-memory {cid: writer} dict to track clients.
- Stores 32-byte public keys briefly and just forwards framed packets to everyone else. No database, no files—relay is deliberately dumb/untrusted.

client.py

- On connect it sends its static public key, then reads the peer key. Prints "Identity" (my fingerprint) and "Key Exchange" (their FP).
- After that, a send/recv loop encrypts/decrypts via the ratchet session. If MAC fails, it bails on that packet.

tests & helper scripts

- test_env.py, test_e2ee.py: ~5 pytest cases to sanity-check crypto pieces (key sizes, MAC verify, ratchet advancing, etc).
- tamper_relay.py flips one byte in each ciphertext, forcing MAC errors to prove integrity actually works.
- Forward-secrecy replay scripts were half-done; I parked that demo for later (see Section 10), mostly ran out of time and it got messy.

7. Results (Evidence tied to Appendices)

Confidentiality
Figure 1 (Appendix) shows a Wireshark dump on TCP port 9000 where everything on the wire is just ciphertext—basically hex junk, no readable strings. Evidence: screenshots/pcap_ciphertext.pcap + screenshot(Appendix Figure 1).

Integrity
Figure 2 (Appendix): when I flip a single byte in tamper_relay.py, the receiving client throws a NaCl MAC verification error (so the packet gets rejected). Evidence: Appendix Figure 2.

Availability
Figure 3a (Appendix): The relay keeps running even with 0 peers connected—logs literally say “0 peers” but it doesn’t crash.
Figure 3b (Appendix): After Bob reconnects, frames start forwarding again normally. Evidence: Screenshot Figure 3b.

Authentication
Figure 4 (Appendix): Both clients print their own fingerprint and the peer’s. Alice’s view of Bob’s FP equals Bob’s self FP (and vice-versa), so the manual check works. Evidence: Figure 4 appendix.

(Forward secrecy live replay demo is missing; see Section 10 for why—I ran out of time and it got messy.)

8. Issues, Fixes & Professional / Ethical Bits

Crypto correctness
I initially messed up HKDF parameters (salt/info didn’t match between sides), which caused early decrypt fails. Wrote a couple of tiny unit tests and standardised those values, then it started working properly.

Asyncio messup

The relay would sometimes crash when a client dropped halfway through a broadcast. I wrapped the writes in try/except and added an asyncio.Lock so the shared client map doesn’t get corrupted. After that, no more random tracebacks.

Tamper vs handshake frames
My tamper script was flipping *every* frame, which nuked the handshake since those 32-byte pubkey frames must stay intact. I changed it to only mess with “application” ciphertext frames, not the initial key exchange.

Ethics & professional practice
I avoided logging plaintext anywhere, and all crypto happens client-side (relay sees ciphertext only). I cited the libs/papers I relied on.

I took AI help for debugging and for understanding the syntax and a few concepts.

9. Strengths, Weaknesses & Trade-offs

Strengths

- Small, pretty clear codebase but still shows CIA + manual auth properly.
- Each property is backed by concrete stuff (PCAP dump, MAC fail screenshot, relay logs), not just claims.
- Basic unit tests around the crypto bits and the ratchet so I knew it wasn’t totally broken.

Weaknesses / Residual risks

- No store-and-forward: if a peer is offline, those messages are just gone forever.
- Static keys aren’t persisted, so smooth reconnects + a proper forward-secrecy replay demo was not possible.
- No GUI—Rich TUI could have been more interactable.
- Metadata (packet length/timing) still leaks, so an attacker can maybe infer patterns even if content stays secret.
- Only two clients can chat with each other at once adding extra clients would need fresh handshakes and new session management which I didn’t implement.

Trade-offs
I deliberately chose “keep it minimal and understandable” over “production features”. That means less complexity to reason about crypto, but also missing

stuff (e.g. message queues, nicer UX). which was in accordance with the 30 hour time frame.

10. <u>Forward Secrecy Note (why the live replay demo is missing)</u>

I *did* wire in per-message key rotation. But proving it with a real replay attack would've needed the clients to stash their static keys on disk and then run a clean re-handshake after restart so I removed the code for that but I did write the code for that. That means extra state and complex code. So I just documented the limitation.

11. <u>Personal Reflection – What Changed for Me</u>

I actually get how the ratchet works now, not just the documentation. I learnt a bunch of asyncio patterns the painful way (locks, handling half-dead sockets, etc.). Also realised "secure" isn't only math—making users compare fingerprints is super annoying but also essential. The hardest bit was balancing scope vs correctness: I had to say "no" to shiny stuff (better GUI, a polished FS replay demo) so the core stayed solid. Time wise I spent ~30 hours: reading/planning (~6h), crypto code + tests (~8h), relay/client (~7h), demos/screenshots (~5h), docs/report (~4h). And linking to CIA was also done. So I learnt how to code while keeping in mind the cybersecurity principles.

12. <u>Conclusion & Future Work</u>

I ended up with a working E2EE chat that actually *shows* the main security props (CIA + auth) and backs them with concrete evidence. So the core goal was met, even if a few edges are rough.

If I keep going, I'd like to:

- **Persist static keys + clean re-handshakes:** so I can properly demo forward secrecy with a replay attack, not just claim it.

- **Encrypted store-and-forward:** to improve availability when peers are offline (right now they just miss stuff).
- **Better UX:** QR codes for fingerprint checks, maybe even a tiny GUI instead of the TUI—less pain for users.
- **Add multi-party support**: maintaining a one ratchet per one peer or switching to a group scheme like MLS

13. Note

Right now there is only 1 to 1 chat possible. The relay practically can forward to more people but my client file only tracks the one peer's public key and one double ratchet state. If another user joins I need to run an independent ratchet session for him and a new handshake UI or redesign to implement a group protocol.

There's more polish possible to this project but the time was the limitation.

References

Perrin, T. (2018) *The Noise Protocol Framework*. Available at: https://noiseprotocol.org/noise.pdf (Accessed 19 July 2025).

Bernstein, D.J. and Lange, T. (n.d.) *NaCl: Networking and Cryptography Library*. Available at: https://nacl.cr.yp.to/(Accessed 16 July 2025).

Marlinspike, M. and Perrin, T. (2016) *The Double Ratchet Algorithm*. Available at: https://signal.org/docs/specifications/doubleratchet/ (Accessed 17 July 2025).

PyNaCl Developers (2024) *PyNaCl Documentation (v1.5.0)*. Available at: https://pynacl.readthedocs.io/en/latest/(Accessed 19 July 2025).

Python Software Foundation (n.d.) *asyncio — Asynchronous I/O*. In: *Python 3.x Documentation*. Available at: https://docs.python.org/3/library/asyncio.html(Accessed 20 July 2025).

Appendix

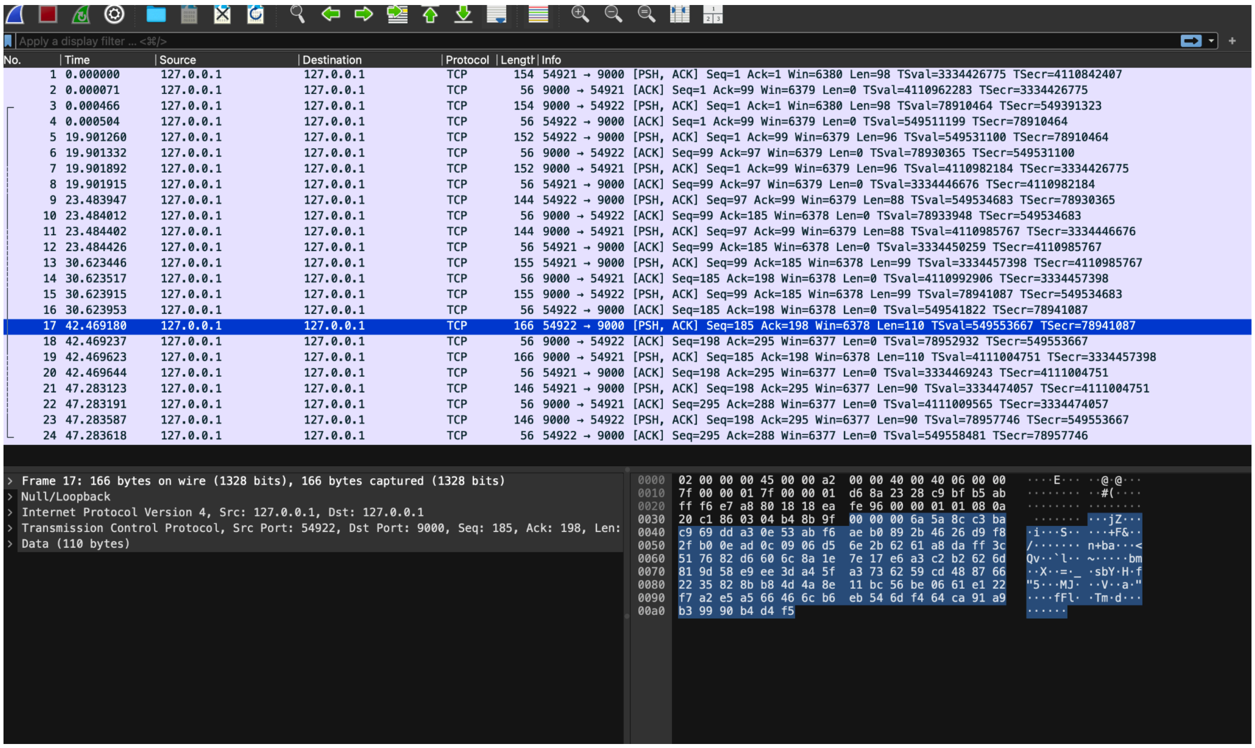


Figure 1

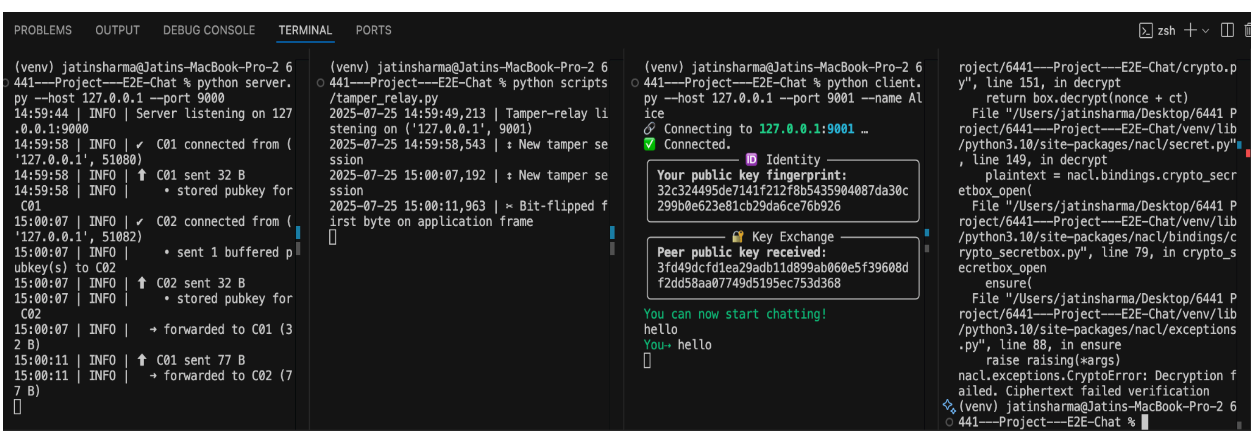


Figure 2

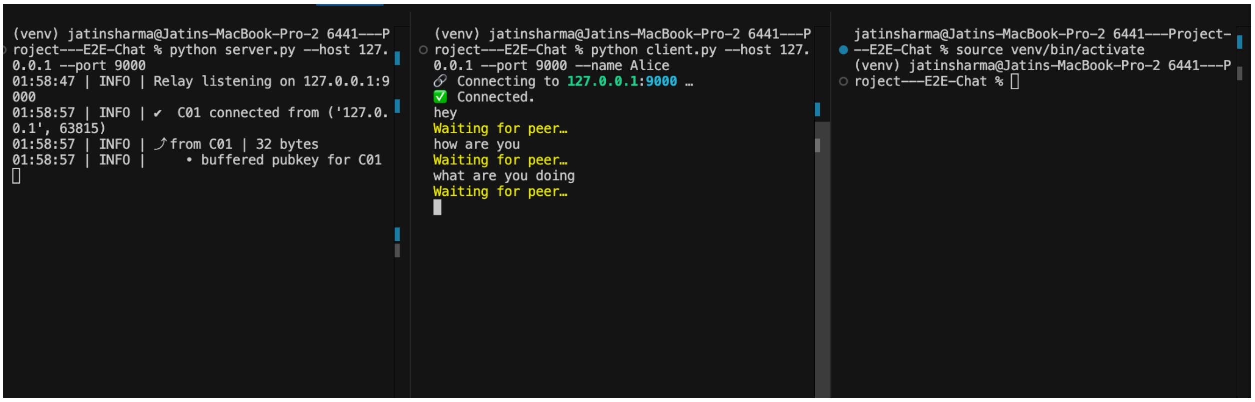


Figure 3a

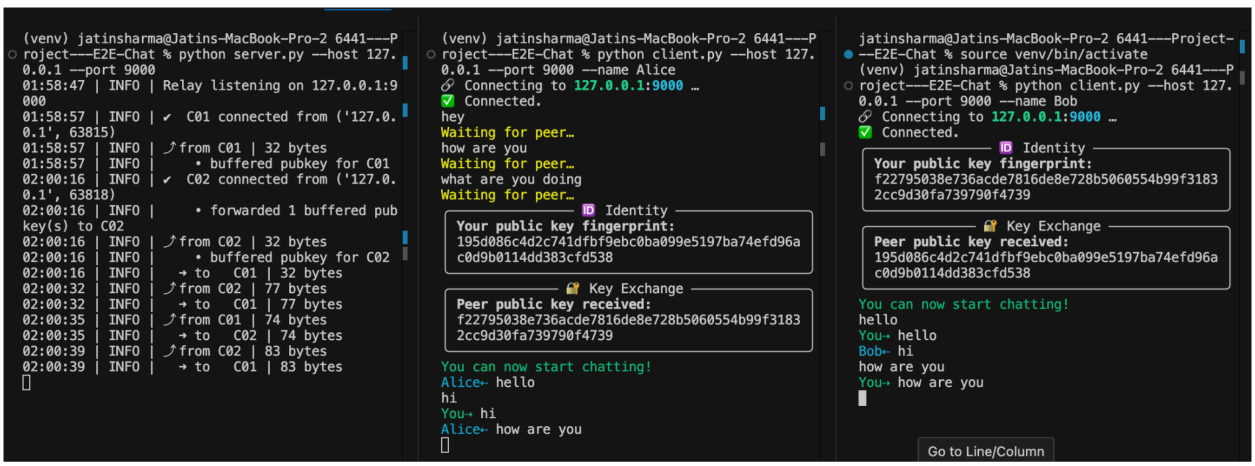


Figure 3b

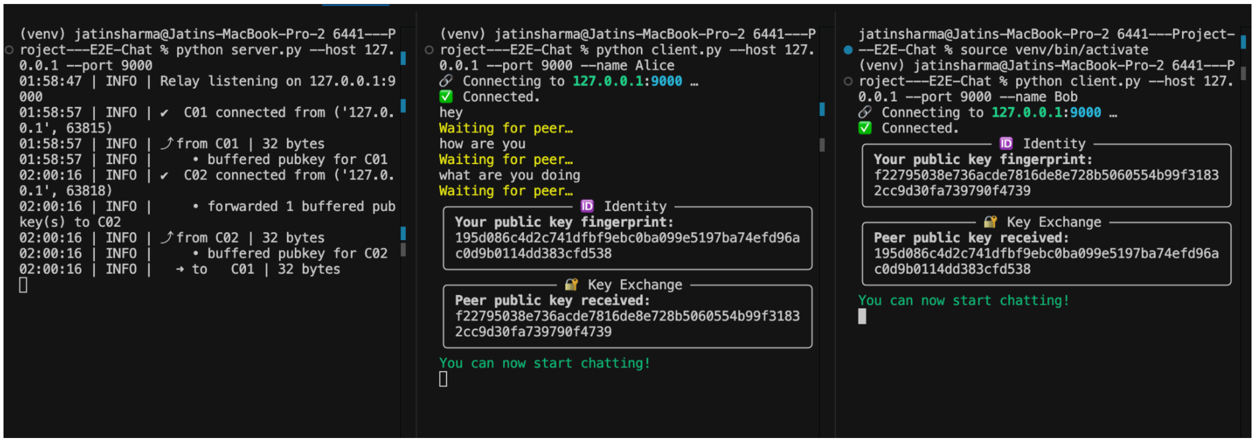


Figure 4

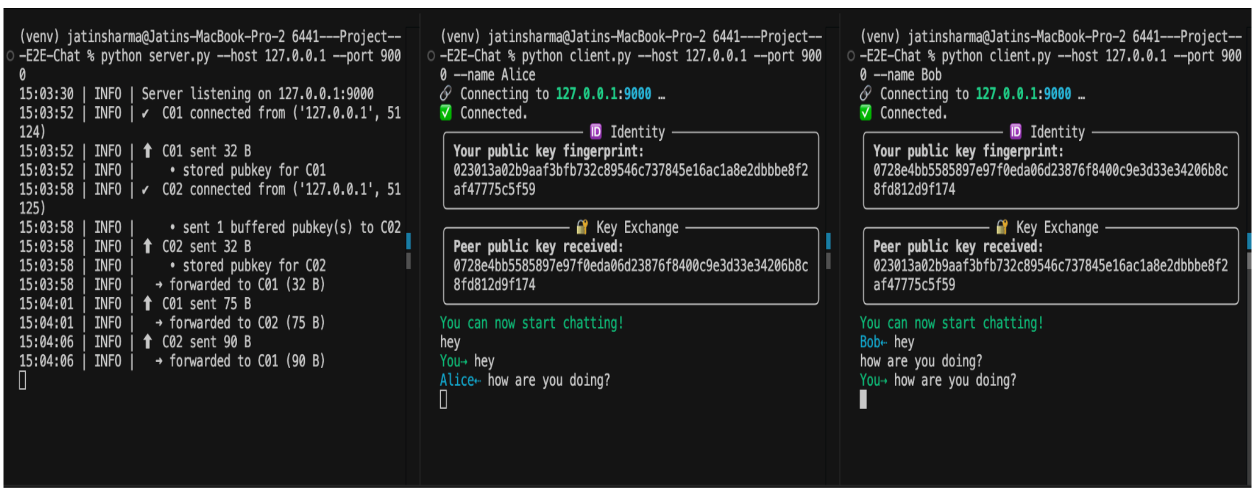
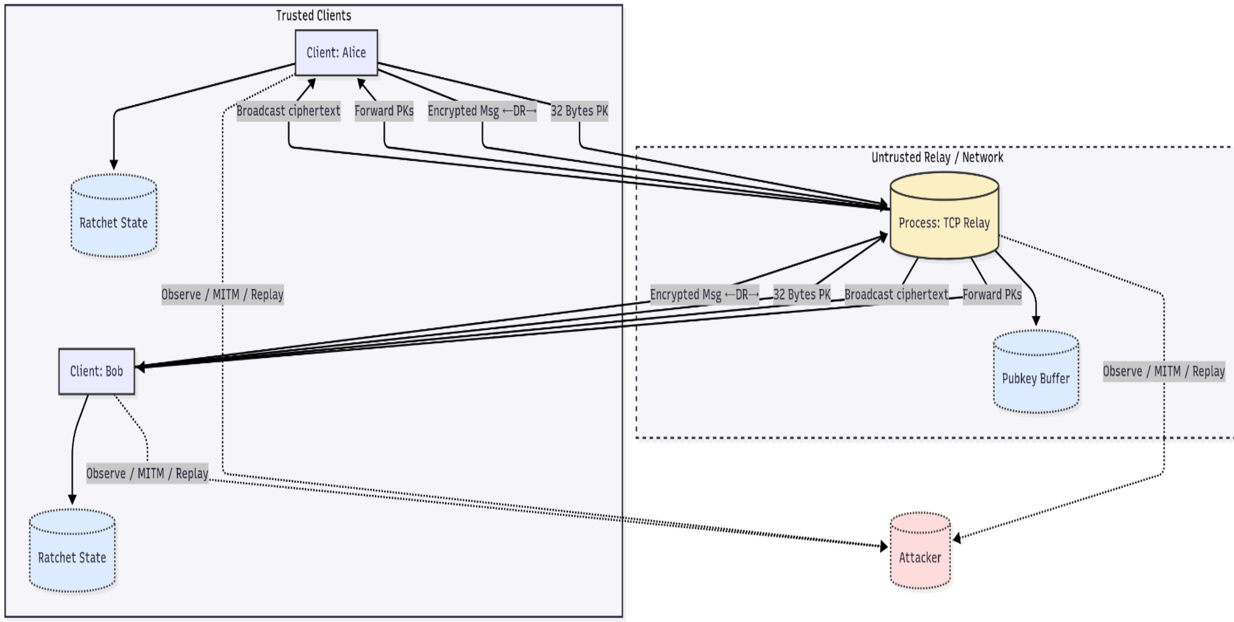


Figure 5 (working model of the E2EE chat)



This is threat_model.png for the chat model

Risk Table

(Quick map of what could go wrong, what I actually did about it, and what's still left as risk.)

Threat (what could go wrong)	Mitigation I put in	Residual / still a problem
MITM during key-exchange	X25519 DH + **manual** fingerprint compare.	If we skip or mis-read the hex, a MITM could sneak in. Human error, basically.
Ciphertext tampering (bit-flip etc.)	XChaCha20-Poly1305 (NaCl SecretBox) MAC on every msg	Attacker can still spam bad frames to trigger MAC errors (DoS-ish), no plaintext leaks tho.
Replay of old ciphertext	I implemented double Ratchet	Cannot write the full working usage as it required significant refactoring
Stolen device / state leakage	Ratchet state only in RAM, keys rotate per message	If someone grabs the device while it's active, they may see future msgs until next ratchet step. Past msgs safe.
Offline peer ⇒ message loss (availability)	Intentionally no store-and-forward in relay	Anything sent while peer is offline is gone forever. I accept that as a trade-off.
flooding	Super small asyncio server	No rate limit or auth, so a bored attacker could just flood messages/frames.
Only 1:1 session supported right now	Kept it simple: one ratchet per peer to focus on core ideas	Adding more users needs new handshakes & per-peer state (or MLS). Not done yet.

Risk_table for my chat model