EPIC 1bit2bit – Project Document

Team 2, Aaron O'Doherty, James Hayes & Skye Fitzpatrick

Submission for CS4436, 3rd June 2026

Table of Contents

Project Overview

Group name: 1bit2qbit

Project URL: https://1bit2qbit.theburkenator.com

- Backend URL: https://1bit2qbit.theburkenator.com/backend/
- Verification page: https://1bit2qbit.theburkenator.com/verify/

GitHub Repository: https://github.com/meltedelement/EPIC-1bit2qbit/

Name + student number	% cont-ribution	Project work undertaken
Aaron O'Doherty 24424048	33%	• Cryptographic Functions for Extended Triple Diffie Hellman (X3DH) and Double Ratchet • Argon2id Server-Side Password Hashing • C++ TUI + Client • C++ Report • Cryptography Report • Cross Platform Fixes
James Hayes 24402796	33%	• Networking TLS logic • C++ TUI work • JSON IPC layer • Local encrypted MessageStore and TOFU certificate pinning • Security hardening • Cross platform fixes
Skye Fitzpatrick 24402095	33%	• Server run script for backend and web application • Server backend message handlers and • Nginx reverse proxy config and security headers • Offline message queue and TTL cleanup daemon • Client key manager generation and encryption at rest. • Client python subprocess handler for communication between crypto and C++ client • Solidity contract, deployment, and testing with NatSpec documentation throughout • Python server message batcher and daemon • Web application and typescript verifier with local proof and Merkle root rebuilding • Whiteboarding and diagrams

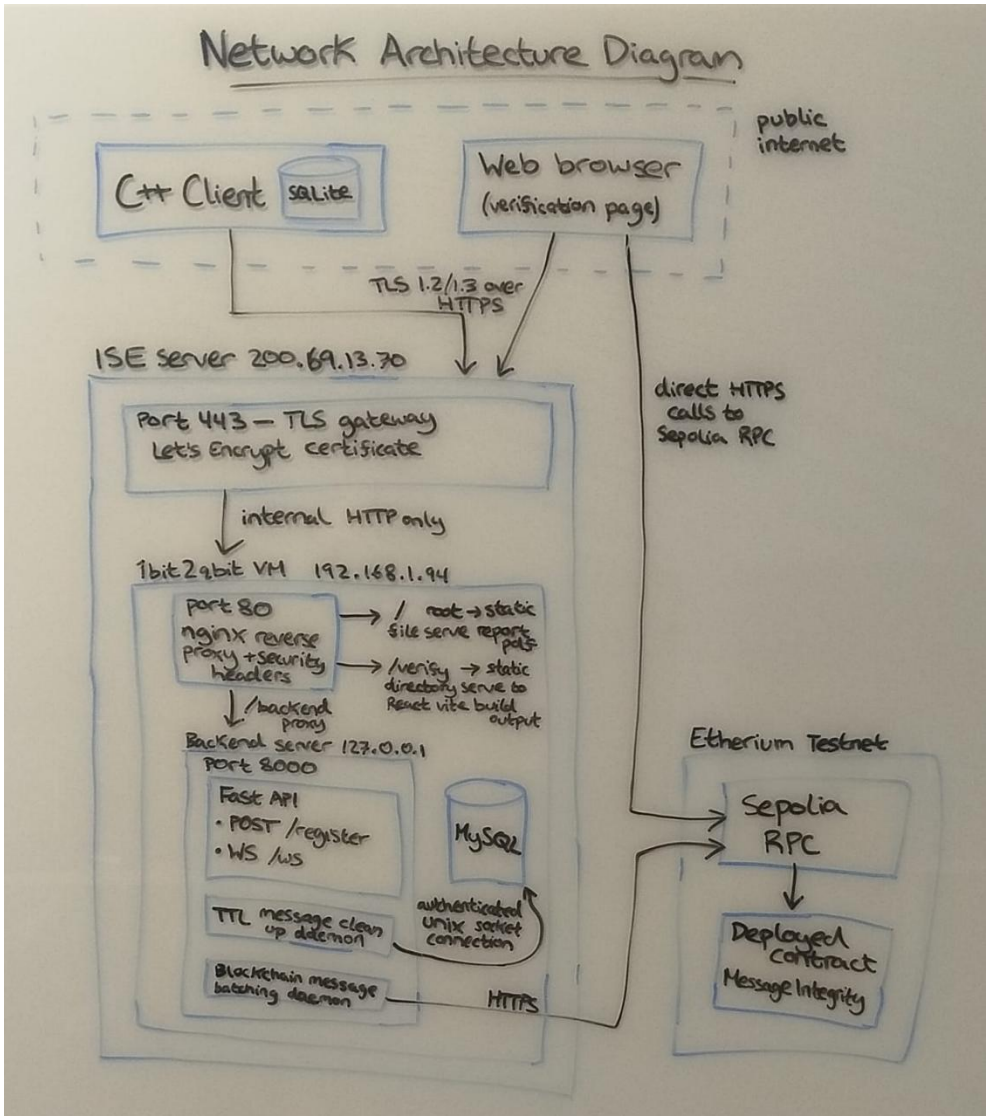
Tech stack used

We write this project primarily in C++ and Python. Our server is a FastAPI server in Python that handles signup requests (POST /register over HTTPS), WebSocket connections, and message routing. Our client is a TUI written in C++ using FTXUI; it uses

raw OpenSSL for TLS with manual WebSocket framing, and talks to a Python subprocess over JSON IPC to perform all cryptography operations. Our smart contract on Sepolia provides message verification through a Merkle-tree batching service.

Computer Networks & Cybersecurity

Network Architecture Diagram



Our network architecture is a mainly C++ client talking to a Python server. We took inspiration from the Signal protocol, only /register runs over HTTPS, and after that the session is established and maintained as a single long-running WebSocket connection. Because of this we use no bearer tokens or session cookies. In its first WebSocket frame after the upgrade, the client authenticates itself, and the connection stays active and authenticated until the user disconnects and closes their socket.

The client and server have a clean split of responsibilities. The C++ application handles all of the networking, while every cryptographic operation is delegated to a Python

subprocess over JSON IPC. C++ deals solely with data on the wire and Python deals solely with secrets. Neither side crosses into the other's role.

The server routes messages without reading their contents in the inner wrapper, because they are end-to-end encrypted. To find the recipient it reads their username from the outer wrapper and forwards the message to their active WebSocket. If the recipient is offline the message is instead placed in a delivery queue that holds messages for 30 days; if they come back online within that window they receive all of their queued messages on login. Every message is also recorded on the blockchain in its final state for tamper evidence.

The server reaches MySQL over a local Unix socket using OS-level auth, (this is why we have the boost library included, it handles no external websocket logic) the blockchain batching service connects to the Ethereum Sepolia testnet over an RPC endpoint.

Network Coding & Socket Programming

On the client side the C++ application performs all of the networking by hand. We resolve the server's hostname to its endpoints, open a raw TCP socket and connect to it, then negotiate TLS over that socket with OpenSSL. Once the encrypted channel is up we manually perform the WebSocket upgrade handshake, and from then on we encode and decode WebSocket frames ourselves over the encrypted socket.

Because we work at this level rather than using a high-level HTTP/WebSocket library, we handle the socket details directly: buffered reads and writes, partial reads and writes, and checking the return values of the socket and SSL calls.

TLS Handshake & Certificate Verification

All of our transport implementation is built directly on OpenSSL rather than a higher-level wrapper, which gives us full control over certificate verification. We are pinned to TLS 1.3 to stop a client from falling back to a weaker protocol version. During the handshake the client sets the Server Name Indication and binds the expected hostname into the SSL session. The certificate chain is verified against the system's trusted store, which works on Fedora, Windows, Ubuntu and should work on most other Linux distributions.

On top of certificate chain validation we additionally use trust-on-first-use (TOFU) verification. On the first connection to the server the client records the certificate's SHA-256 fingerprint and stores it; on later connections it verifies the presented certificate against this stored fingerprint, and any mismatch aborts the connection as a possible MITM attack. This stored fingerprint is persistent across restarts..

Authentication & Authorization

We use two different secrets for authentication: a password for authenticating to the server, and a PIN for unlocking a local DEK (data encryption key) that encrypts data at rest. The PIN is never sent over the wire, so the server can never decrypt the client's messages at rest even if it had access to them. Server-side passwords are stored with Argon2id, following the parameters established in RFC 9106.

Authorization is tied to the connection itself. Because the session is the WebSocket connection, a user can only act once their login frame has been accepted. The server keeps a registry of active sessions and rejects a second connection for an already-logged-in account, so an account cannot be hijacked into two concurrent sessions.

Secure Coding & Input Validation

Every WebSocket frame the server accepts is validated before it is acted on. Frames are parsed and schema-checked with Pydantic, where individual fields carry their own constraints: usernames are length-capped and pattern-matched against an allowed character set, and message identifiers are length-bounded. Any frame larger than 16 KB is rejected and the connection closed (code 1009) before parsing, and a connection that sends five consecutive invalid frames is closed (code 4003). Each failure condition has its own distinct close code: 4000 for a malformed login frame, 4001 for failed authentication, 4002 for a duplicate connection, 4003 for too many invalid frames, and 4004 for rate limiting, so the client is told no more than it needs to be.

Authentication is additionally protected against brute force by an IP-based rate limiter: once an address exceeds the configured number of failed login attempts within the window, further attempts are blocked with a retry-after period (code 4004).

For database access we use the SQLAlchemy ORM with parameterised queries throughout, so user-supplied values are never concatenated into SQL and injection is not possible. Sensitive data is kept to a minimum on the server: it stores only ciphertext and Argon2id password hashes until the blockchain batcher puts them on the blockchain, at which point they are deleted. We never store plaintext messages or passwords, and the at-rest DEK never leaves the client.

Penetration Testing & Security Assessment

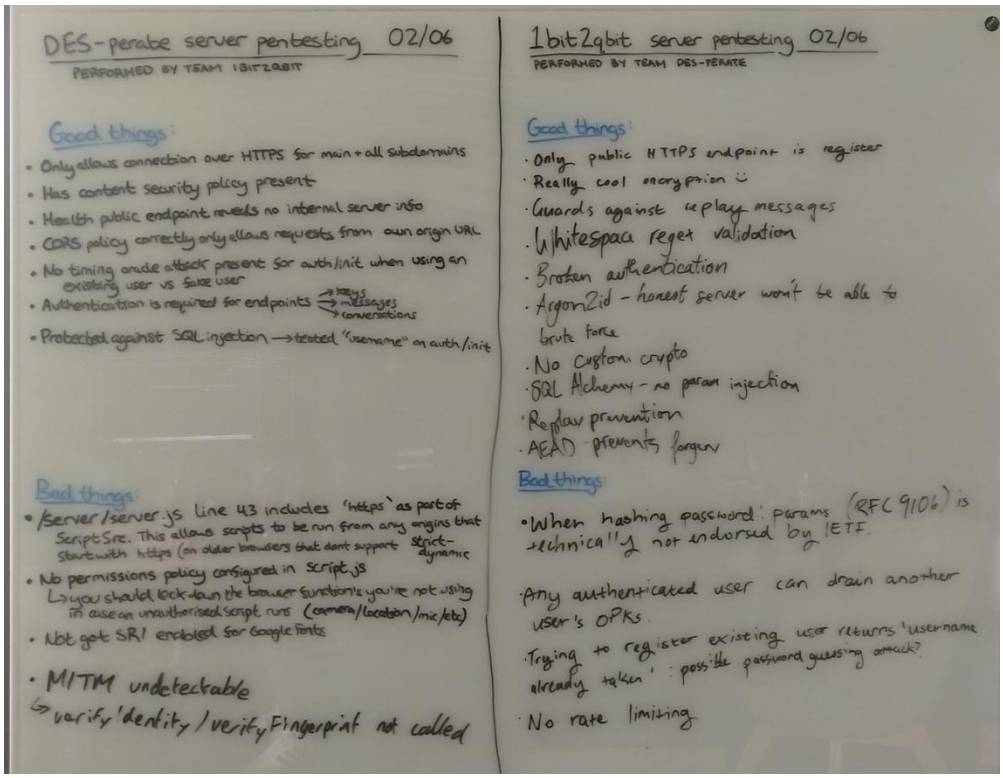
We assessed the system in three ways. First, a day-long reciprocal penetration test where we swapped deployments with team DES-perate and attacked each other's systems using open-source tools, AI assistance, and our own exploits. Second, a full review of the codebase against a defined threat model: an attacker with full knowledge of the source, and a malicious or compromised server (including the TLS-terminating university gateway). Third, a set of external automated scanners. All of the tool output is under docs/security-tests/.

The external scans on 02 June 2026 found no significant issues:

- Qualys SSL Labs, A+. RSA-4096/SHA-256, TLS 1.3 and 1.2 only (1.1, 1.0 and SSL disabled), forward secrecy, HSTS, and no exposure to BEAST, POODLE or similar attacks.
- Security Headers (Snyk), A+. CSP, HSTS, X-Frame-Options, X-Content-Type-Options, Referrer-Policy and Permissions-Policy all set.
- Probely (Snyk API & Web), 0 findings.
- Pentest-Tools network scan, Informational only, 0 risk, with ports 80 and 443 open.
- Tenable Nessus, 0 critical and 0 high, 1 medium: an nginx SSL upstream-injection CVE on the gateway (1.24.0), which we addressed by patching nginx.

The internal review surfaced several high and medium findings, all of which we fixed. The login password was being reused as the local key passphrase, so we split it into a separate PIN and server password. Identity-key pinning now compares the pinned key on every first message. The server certificate pin is persisted across restarts. Double Ratchet state is encrypted at rest under the DEK. We added rate limiting on both login and registration.

Several attacks were tested and the system held up against all of them: SQL injection (blocked by the parameterised ORM), TLS downgrade (refused, since we pin to 1.3), man-in-the-middle via a swapped certificate (caught by the server cert pin), malformed and oversized WebSocket frames (rejected by Pydantic validation, the 16 KB cap and the 5-strike close), username enumeration (defeated by dummy-hash timing), and sender spoofing (the sender identity comes from the authenticated session, not the client).



C++ Programming

C++ Code Component Structure

The client is organized under src/ into six subdirectories, each with a focused concern:

- **main.cpp** – entry point that constructs and runs the Client
- **client/**
  - Client.h/.cpp: the main orchestrator handling login, registration, send/ receive, and session management
- **connection/**
  - Connection.h / .cpp: TCP → TLS → WebSocket pipeline with read/ping threads
  - TlsContext.h / .cpp: SSL_CTX ownership and CA + TOFU cert verification
- **crypto/**
  - CryptoProxy.h /.cpp: IPC bridge to a Python subprocess for X3DH and Double Ratchet operations
- **messaging/**
  - Message.h: Message struct, MessageType enum, RatchetHeader, X3DHHeader
  - Conversation.h / .cpp: Per-peer message list + ratchet state + pinned identity key
  - MessageStore.h / .cpp: SQLite-backed persistence (messages, keys, TOFU pins)
- **models/**
  - User.h / .cpp: User struct (username + TOFU identity key fingerprint)
- **ui/**
  - App.h / .cpp: all FTXUI terminal UI components, event handlers, and rendering

Class Structure

Class	Responsibility
Client	Owns and wires all subsystems and the full session lifecycle.
Connection	TCP connect -> TLS Handshake -> WebSocket Upgrade -> Framed Input/Output
TlsContext	Owns SSL_CTX. Enforces TLS 1.3 with a CA (Certificate Authority) chain and TOFU (Trust on First Use) certificate pin.
CryptoProxy	Spawns Python subprocesses for cryptographic functions. Routes JSON-RPC calls over stdin/stout pipes

Conversation	Peer to Peer session container. Holds the ordered message list, the Double Ratchet state blob, X3DH associated data, and TOFU-pinned identity key for the other peer in question.
MessageStore	SQLite RAII wrapper for messages, ratchet states, Data Encryptyion Key, and TOFU pins
App	All FTXUI components, event handlers, and rendering. Fires callbacks to Client

Memory Management

- Client owns all subsystems via std::unique_ptr<Connection>, <CryptoProxy>, <MessageStore>, and <App> These lifetimes are automatic and explicit.
- MessageStore wraps a raw sqlite3* with RAII: opened in a constructor, closed in a destructor, never exposed outside the class.
- Connection and Client use local RAII guard structs (e.g. struct Guard { X509* c; ~Guard(){X509_free(c);} }) for OpenSSL C-API handles.
- All resource-owning classes delete their copy constructor and copy-assignment operator to prevent accidental duplication.
- No raw new or delete anywhere, all dynamic allocation goes through STL containers or smart pointers.

STL Containers and Algorithms

Containers:

- std::vector - ordered message list in Conversation
- std::map<std::string, Conversation> - live sessions keyed by peer in Client
- std::map<std::string, std::vector<std::string>> - pending outbound messages awaiting key bundles
- std::optional<T> - all MessageStore load methods, Message::target_id, X3DHHeader::pre_key
- std::atomic<bool/int> - thread-safe flags in Client (quit, lockout) and Connection (running, connected)

Algorithms:

- std::remove_if - erase-remove idiom in Conversation::remove_message
- std::find_if - locate a conversation by peer name in Client

- std::transform -case folding in the WebSocket header parser
- std::min - exponential backoff cap in Client::reconnect_loop
- std::distance: iterator-to-index when selecting a new conversation

Lambdas:

- [this] captures wiring UI actions to Client methods
- [id](const Message&) predicates for std::remove_if
- [&] closures in FTXUI renderers

Build and Run Instructions

The client consists of two components that run together:

- **C++ binary** (`epic-client`) — terminal UI, networking, state management
- **Python subprocess** (`subprocess_handler.py`) — all cryptographic operations (X3DH, Double Ratchet, DEK management)

C++ binary — Windows (MSYS2)

MinGW is a port of the GCC compiler to Windows. MSYS2 is the recommended distribution — it ships MinGW-w64 with a package manager (`pacman`) that provides pre-built Windows-native versions of all required libraries.

The MSVC OpenSSL installer (`OpenSSL-Win64`) is **not compatible** with MinGW — its `lib/` only contains MSVC `.lib` files. Use the MSYS2 packages below instead.

1. Install MSYS2

```
winget install MSYS2.MSYS2
```

2. Open the MSYS2 MinGW x64 shell from the Start menu and install the toolchain and dependencies:

```
pacman -Syu
pacman -S mingw-w64-x86_64-gcc \
          mingw-w64-x86_64-cmake \
          mingw-w64-x86_64-ninja \
          mingw-w64-x86_64-openssl \
          mingw-w64-x86_64-boost \
          mingw-w64-x86_64-sqlite3
```

3. Configure (first time only, or after changing `CMakeLists.txt`):

```
cd /c/path/to/EPIC-1bit2qbit/client
cmake --preset windows-msys2
```

4. Build (run this after every source change):

```
cmake --build --preset windows-msys2
```

The binary is written to `build/epic-client.exe`.

5. Copy runtime DLLs (first time only)

The exe depends on MinGW runtime DLLs that must sit next to it so Windows finds the correct versions regardless of what else is in PATH:

```
$src = "C:\msys64\mingw64\bin"
$dst = "C:\path\to\EPIC-1bit2qbit\client\build"
@(
    "libgcc_s_seh-1.dll",
    "libstdc++-6.dll",
    "libwinpthread-1.dll",
    "libcrypto-3-x64.dll",
    "libssl-3-x64.dll",
    "libsqlite3-0.dll"
) | ForEach-Object { Copy-Item "$src\$_" "$dst\$_" -Force }
```

Re-run this step after upgrading MSYS2 packages.

Clean rebuild (if you need to start from scratch):

```
cmake --build --preset windows-msys2 --clean-first
```

C++ binary — Linux

```
sudo apt install build-essential cmake ninja-build libssl-dev
libboost-dev libsqlite3-dev
cd client
cmake --preset linux-debug
cmake --build --preset linux-debug
```

Python crypto subprocess

Requires Python 3.11+.

```
cd client
python -m venv .venv

# Windows
.venv\Scripts\activate
# Linux
source .venv/bin/activate

pip install -e ".[dev]"
```

Run the test suite:

```
pytest
```

Dependency summary

C++ (CMake)

Library	Version	Source
FTXUI	5.0	FetchContent (if not found locally)
nlohmann_json	3.11	FetchContent (if not found locally)
CLI11	2.4	FetchContent (if not found locally)
OpenSSL	any	System (MSYS2 or apt)
Boost	1.74+	System (MSYS2 or apt) — header-only, Asio
SQLite3	any	System (MSYS2 or apt)

Python

Package	Version
doubleratchet	≥1.0.0
x3dh	≥1.3.0
cryptography	≥43
argon2-cffi	≥23.1

Cryptography

Threat Model

Adversary	Security Properties Held
Passive network attacker	TLS 1.3 and our AEAD scheme (AES-256-GCM) ensure ciphertext confidentiality, a passive network attacker cannot break authenticity or integrity without becoming active.
Active network attacker	Confidentiality is held the same as above, with integrity being held by AEAD. Authenticity is guaranteed in an established session by the AEAD tag, however if an attacker hijacks the initial key bundle exchange, they can pose as the recipient, however this requires bypassing the TLS CA verification to conduct this attack.
Honest-but-curious server	Confidentiality is held as the server only ever sees and stores the ciphertext of any given message. An honest server will relay these messages without altercation, verifying integrity, but any modification would fail the GCM tag. AEAD verifies authenticity, as our server never sees any client's private keys.
Fully compromised server	Confidentiality of past messages/existing message sessions is maintained by the double ratchet and AEAD scheme, provided it's a previously verified conversation. However, due to our TOFU key exchange, a server *can* compromise an initial shared secret derivation by providing malicious substitute keys to a user, posing as the recipient, meaning confidentiality is not verified on any new conversations on the compromised server. Similarly, integrity can be forged on an initial key exchange. A compromised server can tamper with a message, however this will be caught by AES-GCM and dropped. IT cannot successfully tamper a message without failing authenticity, but it can block all traffic.
Fully compromised client	If a client's device is compromised with a session open, nothing holds, an attacker bypasses all security checks on the client and can begin messaging on behalf of them. If a session is not active, to emulate a victim the attacker needs both their authentication password, and their PIN to access the unencrypted DEK. The authentication password is only stored in C++ heap memory, never anywhere concrete, as is the unencrypted DEK when a session is open.

Cryptographic Primitives and Parameter Justification

Authenticated Encryption with Associated Data (AEAD)

Our algorithm for AEAD was AES-256 in Galois Counter Mode (AES-256-GCM)[1].
We chose this algorithm as GCM is the recommended block cipher mode by NIST publication 800-38D.

We used a random, 96-bit initialisation vector/nonce, which is the exact size recommended by NIST for which a random IV is permissible.

Nonce reuse is catastrophic within GCM, but our AEAD resides within a double ratchet, which uses a new key for each message, meaning both the key and the nonce would have to match for there to be a collision, the chance of which is negligible.

A 256-bit key size keeps us safe from Quantum adversaries, as Grover's algorithm effectively halves the key size due to its $\sqrt{N}$ optimisation (2^256 -> 2^128 bits of entropy)

Key Establishment

We used X3DH [2] (Extended Triple Diffie Hellman) for our initial key agreement, which then feeds into the Double Ratchet [3] session, both a part of the signal protocol.

The Double Ratchet is the gold standard for peer-to-peer messaging, due to its new ephemeral key exchange with each message chain. It is a combination of both a Diffie Hellman ratchet and a symmetric key ratchet.

This provides us with perfect forward secrecy, as the compromise of long-term keys will not expose any of these ephemeral keys.

The Double Ratchet also provides post-compromise security, as if a session key is compromised, security recovers upon another step of the asymmetric, Diffie Hellman ratchet.

We have rate limiting in place to prevent users from requesting key bundles multiple times from the same other user, limited to once per hour. This should only be done once as an initial conversation establishment anyway.

Server-Side Password Hashing

We used Argon2id with RFC-9106 [4] recommended minimum parameters for a memory-constrained environment.

These are a time-cost (iteration) count of 3 with 64 MiB of RAM, with a parallelism of 4.

We prevent against timing attacks by computing a Dummy Hash on a password "dummy" if a username provided is not found, rather than instantly returning. This means that for a malicious actor testing different login combinations, there is no difference in timing between when a user exists or doesn't in our database.

Key Derivation

Our Key Derivation function used is HKDF-SHA-256.

SHA-256, the hash function, is recommended by the signal protocol for their double ratchet, and approved by the NIST as "acceptable for all hash-function applications" [5]

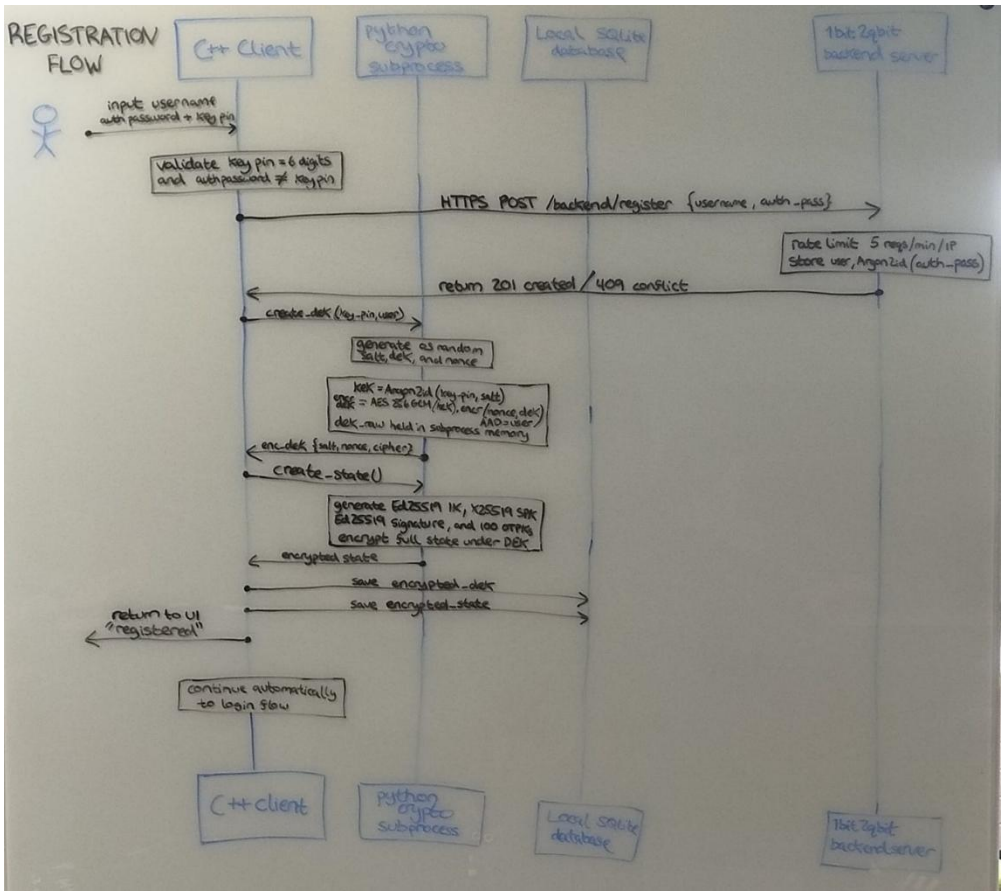
HKDF is recommended by RFC 5869[6].

Key Storage at Rest

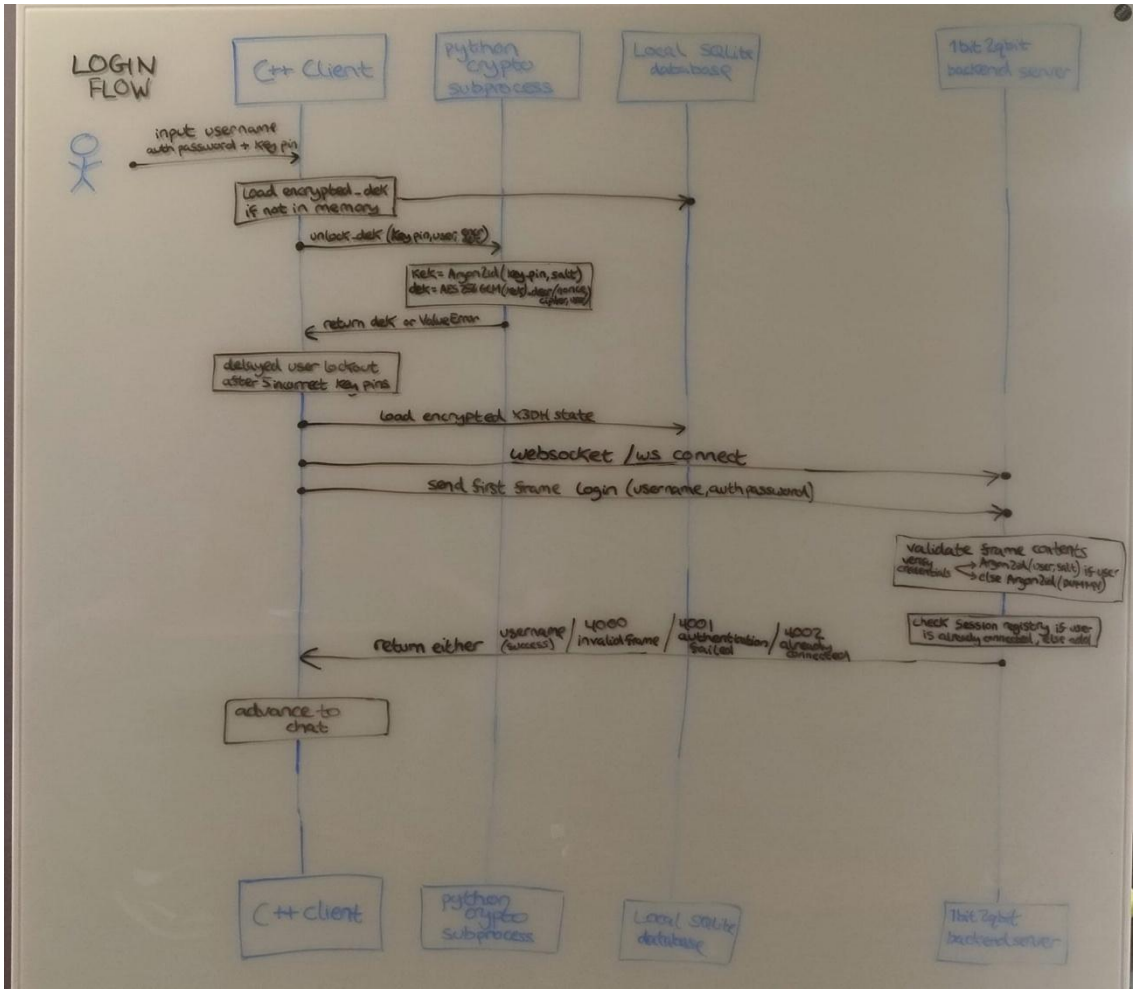
We used Argon2id with the RFC-9106 recommendation for key derivation for hard-drive encryption. (t=1, 6GiB RAM, p=4)

Construction Walkthrough Diagrams

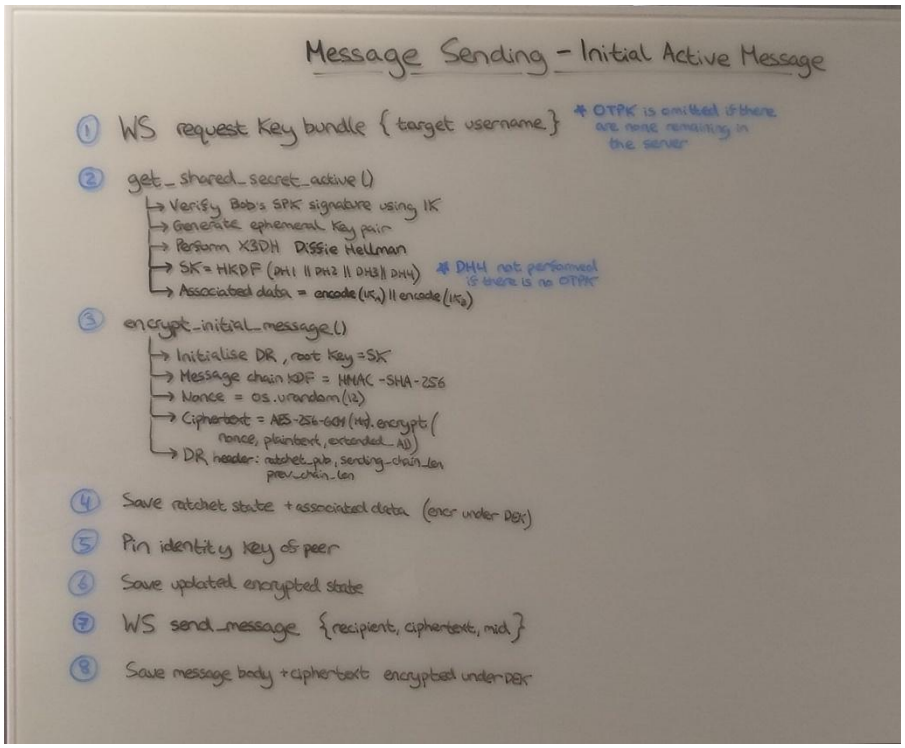
Registration



Login and authentication



Message Sending



Known Limitations

Post-Quantum Security: both X3DH and the Double Ratchet rely on X25519, an elliptical curve (Curve25519) is used for key exchange. A sufficiently powerful quantum computer can derive the initial shared secret derivation and every ratchet step.

We considered implementing ML-KEM for our initial key exchange, (PQXDH signal protocol) which would prevent against "harvest now, decrypt later" attacks as ML-KEM is post-quantum secure, however we decided that it did not fit within the scope of the project because there are no existing vetted libraries that implement this as of now, and we did not deem it necessary to implement our own cryptographic primitives.

ML-KEM alone also does not ensure complete post-quantum security, as the existing double ratchet is still susceptible to an active quantum adversary, breaking post-compromise security.

Given more time, we would implement a fork of the existing python X3DH library used and incorporate ML-KEM into the initial key exchange, to protect our current users from quantum attacks in the future.

Server-Side Argon2id Parameters: Our Argon2id parameters for our server-side password hashing are limited by our virtual machine's limited hardware, meaning the hardness of our hash is less than we would have on a stronger server.

Given another medium on which to host the server, we would increase these parameters to the recommended server memory hardness specifications in RFC 9106, instead of our current memory-constrained recommendation.

OTPK Exhaustion: We stored 100 one-time-pre-keys at a time, meaning if someone were to request over 100 from a particular user before they logged in and refreshed the pool, this means forward secrecy breaks until these are refreshed, as there's no one-time use key used on subsequent messages.

In practice, we are refreshing whenever we use any OTPKs, so as long as the user is active, they'll refresh.

Skipped Key Accumulation: In a similar vein, our max number of skipped keys in our double ratchet is 100. If 100 messages are sent out-of-order, the double ratchet will begin rejecting them.

Ratchet De-Sync: A known limitation of the Signal Protocol's double ratchet, due to the nature of the ratchet, if 2 users' ratchets ever desync, they will be unable to encrypt and decrypt each other's messages. Our double ratchet handles out-of-order messages, this would only happen in the event of an old backup, or a database error causes desync where the ratchet has been updated but the functions dependant on it fail.

Blockchain

Trust Model & Decentralisation

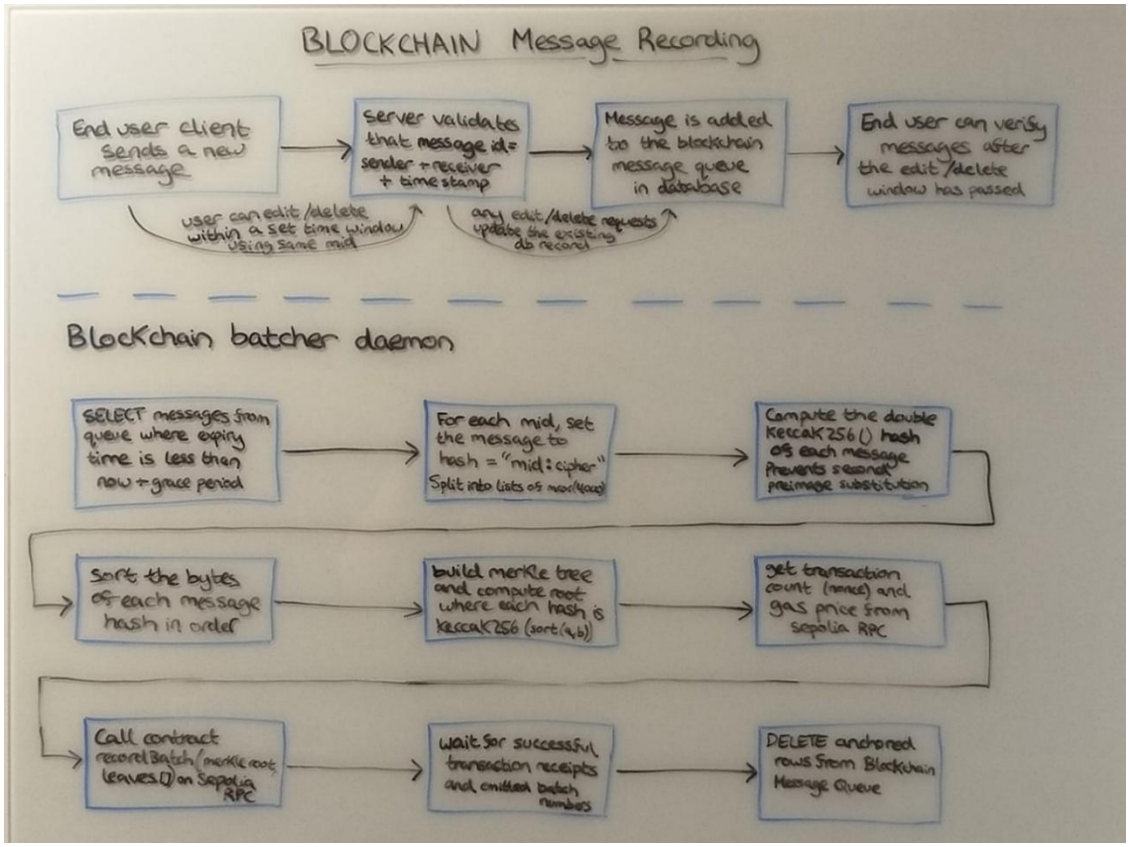
Our integration of blockchain provides tamper evidence through a fully decentralized model, with no data stored on the server necessary for verification. The MessageIntegrity smart contract is deployed once to Sepolia and is append only, meaning that no party including the server can alter records once they are written.

We chose to perform batch writing on the server (enforced via onlyOwner) rather than directly from client applications, so that we could consolidate more messages into one transaction to save costs on gas fees, and also so that transaction signatures on the client would not require the private key of our Contract owner's wallet address.

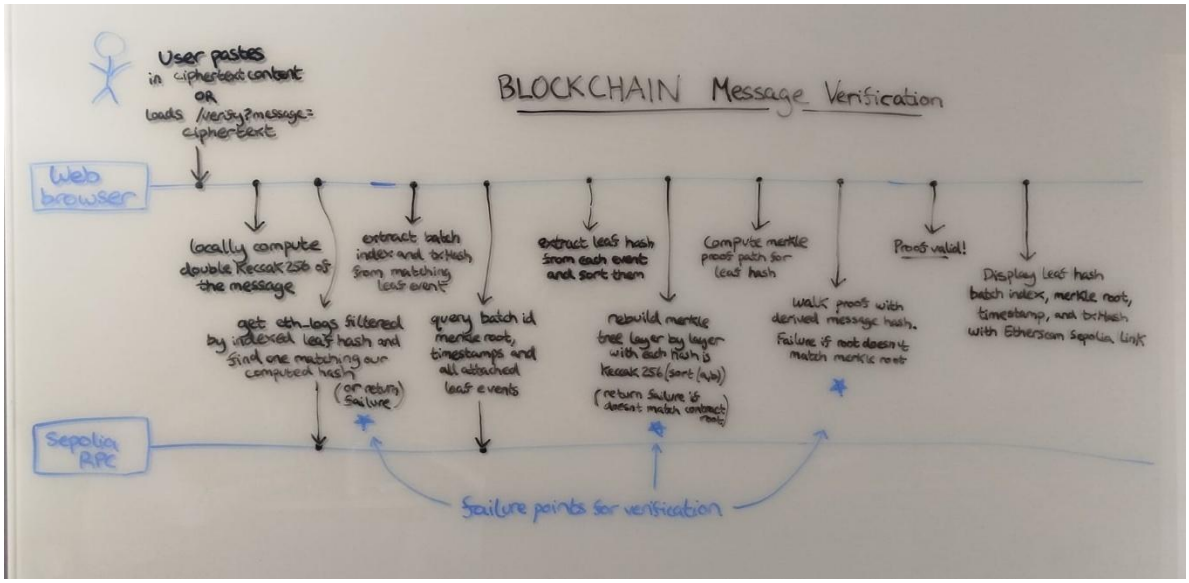
The main contract function, recordBatch, stores the Merkle root and timestamp of a given batch of ciphertext message. Every individual leaf of the Merkle root is emitted as an indexed LeafRecorded event in the transaction logs, which is significantly cheaper in gas than direct data storage on the blockchain.

The webapp verification page requires no contact from the backend server. It is a standalone React and TypeScript page that queries an RPC node directly from the browser using ethers.js. It recomputes the message hash, Merkle tree, and Merkle root proof path all locally, to provide a clear verification pass/fail message to the user.

Message Recording



Message Verification



Key Design Decisions

Leaves are emitted as events, not storage

The leaf hashes that make up the Merkle root are necessary for verification and root path derivation. Emitting them as LeafRecorded events costs around 375 gas per indexed topic, compared to storing a direct value with SSTORE which costs roughly 20,000. Events are queryable via eth_getLogs using indexed topics, so there is no gas cost for data retrieval on the verification page.

Indexed event fields

Solidity caps event types at 3 indexed fields, so for LeafRecorded(batchIndex, leafIndex, leafHash), we have chosen to index both batchIndex and leafHash to allow for faster fetching on the RPC. There is no need to index leafIndex because it is only used for ordering after retrieval, and is not directly searched.

No contract level root verification

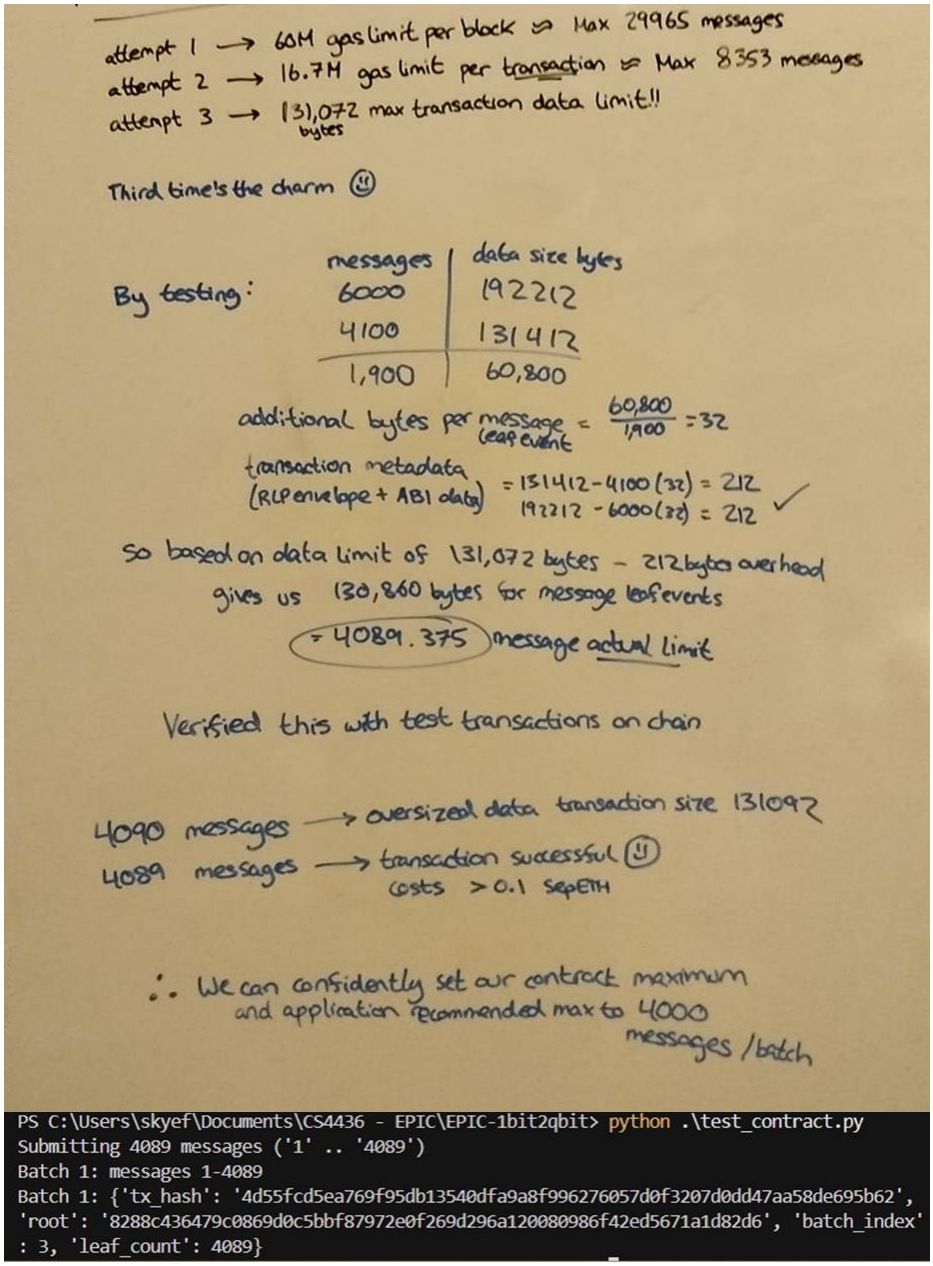
On batch submit, the contract does not verify that the given Merkle root is correct for the list of submitted leaves. Doing this would require O(n log n) keccak256 operations, which is highly gas inefficient. Trust is instead enforced with the onlyOwner guard, so that nobody else can submit messages.

SortPairs Merkle convention

Both the python batcher and TypeScript verifier use the SortPairs convention for building Merkle roots, meaning that adjacent siblings are always sorted before computing the Keccak256 hash. The purpose of including this is to that the verifier does not need directional metadata about siblings in the proof.

Maximum batch size 4000 messages

Our MessageIntegrity contract enforces a hard limit of 4000 messages for one Merkle root. This figure was derived from the maximum transaction data limit of 128KiB, meaning that any leaves emitted beyond 4089 is rejected. We originally based our maximum messages decision off of the gas limit per block, then gas limit transaction, before we empirically determined the real message event limit and confirmed with transactions on the blockchain.




```
PS C:\Users\skyef\Documents\CS4436 - EPIC\EPIC-1bit2qbit> python .\test_contract.py
Submitting 4089 messages ('1' .. '4089')
Batch 1: messages 1-4089
Batch 1: {'tx_hash': '4d55fcd5ea769f95db13540dfa9a8f996276057d0f3207d0dd47aa58de695b62',
'root': '8288c436479c0869d0c5bbf87972e0f269d296a120080986f42ed5671a1d82d6', 'batch_index'
: 3, 'leaf_count': 4089}
```

AI Usage Declaration & Reflection

Group AI Usage Approach & Code Review setup

Our universal approach to prompting was inspired by this seminar by Anthropic, creators of Claude: https://www.youtube.com/watch?v=ysPbXH0LpIE
Claude Code was used for the majority of development.

We implemented an AI-review, then human-review feedback process on pull requests, by automatically getting Github Copilot to review pull requests on creation, and each pull request was reviewed by a separate team member before merge.

We also used an automated Pytest framework to run tests on all our generated code, as well as a C++ client build and push test and hardhat automated testing for blockchain.

Aaron O'Doherty

In line with the seminar linked above, I implemented an iterative prompting strategy. Claude and other LLMs work best the more context they are given. Claude Code is always given the full code repository, so it always has access to the "Dynamic content" that Anthropic believes should be in a prompt structure. I always tried to direct Claude to where it needs to go, to stop it from searching aimlessly for what I have asked for. There was one instance where I used Claude for an incredibly computationally heavy task and had to resort to ChatGPT for a brief period.

Upon reflection, I believe there were many instances where my instructions were too vague, or unclear, forcing Claude to pick up on the scraps and provide it's own context, which was often time-consuming and also incorrect at times, as it was guessing what I wanted to do rather than me fully defining the situation.
Something I will do more in the future is especially in a long, information heavy prompt, to repeat the critical task/instructions that I need.

James Hayes

While prompting I frequently made use of the different effort levels and models, tuning them depending on the task at hand. I used claude code, clearly defining tasks, prompting it to ask clarifying questions, writing it reference documents for complicated tasks and making use of the plan tool for large tasks.

Skye Fitzpatrick

My approach for LLM based code generation is comprehension first, explicit human choice of design decisions second, and code generation as a final step piece by piece with human review. I worked incrementally alongside Claude Code, and reset sessions with fresh context for each modular piece of work I am performing.

I have submitted all my AI logs for this project, with each session clearly labelled as the date followed by branch name.

References

- [1] NIST GCM specification: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf
  [2] Signal Protocol X3DH specification: https://signal.org/docs/specifications/x3dh/
- [3] Signal Protocol Double Ratchet ???https://signal.org/docs/specifications/doubleratchet/?
- [4] RFC 9106, Argon2id specification: https://www.rfc-editor.org/info/rfc9106/
- [5] SHA-256 NIST Approval: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf
- [6] HKDF RFC 5869 Specification: https://www.ietf.org/rfc/rfc5869.txt