

SecureConnect: A Scalable Infrastructure for End-to-End Encrypted Communication

Ronit Sarkar¹, Manish Mahimkar², Atharva Shelke³, Sahilkumar Basude⁴, Prof. Rajni Ratnaparkhi⁵

^{1,2,3,4} Student, Dept. of Computer Engineering, Dilkap College of Engineering, Maharashtra, Raigad, India

⁵ Professor, Dept. of Computer Engineering, Dilkap College of Engineering, Maharashtra, Raigad, India

Abstract - SecureConnect, a very extensively modular full-stack messaging and voice-calling platform, which was designed with a rigid zero-knowledge architecture. To grant the highest level of privacy, our team adopted a client-side only cryptographic design where all sensitive processes such as the RSA-2048 key generation and the AES-256 encryption are done in the browser by the Web Crypto API, so that the customer messages and the raw confidential keys are not disclosed to the server. The system is a layered Model-View-Controller (MVC) architecture, which is based on a Node.js runtime and a thin, 119-line server.js entry point that delegates business logic to special directories.

The backend architecture relies solely on the HTTP framework layer and is built with Express.js, which provides RESTful API routing and a powerful security middleware bundle that incorporates Helmet to manage the Content Security Policy (CSP) and JWT authentication along with a brute-force protection layer via express-rate-limit. On the other hand, the real-time communication layer is driven by pure Node.js and makes use of Socket.IO to support two-way messaging and WebRTC to support peer-to-peer (P2P) and mesh-based group calls. In order to be scalable, we added an optional multi-instance Socket.IO synchronization Redis adapter,

The integrity of the data is ensured by the Sequelize ORM linked with an SQLite3 database, which has a comprehensive schema, which deals with the users, messages, groups, and credit transactions. One of the most important innovations of our project is an integrated credit management system, which makes use of the atomic Sequelize transactions to safely deduct user credits to perform certain operations, like sending a message or connecting the WebRTC calls. Without jeopardizing security, we designed a key synchronization flow to secure the access of the private keys by cross-device access using PBKDF2-derived keys.

Lastly, our team implemented a current DevOps and testing process using Jest and Supertest to validate the API and Playwright to test the end-to-end browser. The app is also streamlined with the configuration of the high-performance cloud on AWS EC2 through GitHub Actions CI/CD, with the PM2 process manager as a constant performance monitor and automatic restart. In conclusion, it can be stated that

SecureConnect manages to prove that advanced digital privacy does not have to be costly and technologically vulnerable.

Key Words: SecureConnect, MVC Architecture, End-to-End Encryption (E2EE), Node.js, Socket.IO, WebRTC, Sequelize ORM, SQLite3, Pay-per-use, Web Crypto API.

1.INTRODUCTION

Our group came up with a complete web application named SecureConnect, which is created in a digital world where privacy and data security are the main priority, and it is implemented through effective communication. It offers a combined solution to end-to-end encrypted (E2EE) messaging and voice call and has a very responsive and user-friendly web interface.

The key to our project is the zero-knowledge architecture, and therefore, all cryptographic functions are performed on the client side only. Our system uses RSA-2048 keys and AES-256 encryption in the browser directly by using the Web Crypto API. This architecture will ensure that our servers are never exposed to plaintext messages, private keys, or raw passwords; they can be nothing more than unsafe relays to encrypted blobs.

In addition to security, we had a new monetization model that was flexible and pay-as-you. SecureConnect has an inbuilt credit management system unlike the traditional subscription-based services which tend to confine a user to strict levels of payments. This system makes use of the atomic database transactions to withdraw credits on certain actions- sending a message, creating a WebRTC call connection, etc.- to be financially transparent and, at the same time, data-intensive.

To make the platform scalable and maintainable, our team has developed the platform on a modular MVC (Model-View-Controller) pattern. We have used Express.js as the HTTP framework layer to address RESTful API routing, security middleware and the core of the system runs on Node.js. This has Socket.IO to direct peer-to-peer audio streams (WebRTC), and Socket.IO to direct peer-to-peer messaging events. Data is operated through the Sequelize ORM over an SQLite3 dialect, an easy to lightweight although powerful relational database framework to manage users, groups, and histories of transactions.

Finally, SecureConnect is the solution that connects the digital privacy and advanced technologies with the accessible and practical communication.

2. Methodology

Our team put together SecureConnect using this modular setup with microservices and MVC patterns. We mixed in serverless stuff from the cloud and made sure to follow strong crypto rules right from the start. It was kind of a layered approach, breaking everything into parts that could work separately but still connect.

Frontend: Our group project frontend is written in a Vanilla ES6 architecture based on Vanilla HTML5, CSS3 and JavaScript, which offers a lightweight and high-performance interface across seven separate pages. We divided the client-side logic in to functional modules such as api.js, a fetch wrapper, used to communicate with the RESTful server, socket.js, which is used to deal with real-time bidirectional events, and webrtc.js, which is used to peer-to-peer and mesh group voice signaling. The crypto.js module, which is based on the Web Crypto API, was used in order to perform all the cryptographic operations, such as RSA-OAEP 2048-bit key generation and AES-GCM 256-bit encryption entirely in the browser of the user, to ensure the environment is zero-knowledge. Such a flexible architecture is facilitated by dedicated UI assistants and core logic scripts such as auth.js and chat.js to make the implementation of a security-enhancing and real-time messaging integration go smoothly.

Backend: SecureConnect has its backend designed in a modular MVC (Model-View-Controller) pattern that runs on the runtime of the Node.js the Express.js framework. The system uses a skinny, 119-line server.js entry point, which transfers business logic to controllers, routes, and specialized middleware directories. Where Express makes use of the HTTP request/response cycle, pure Node.js makes use of real-time capabilities via Socket.IO with a bidirectional messaging protocol and WebRTC signaling and an optional Redis adapter to scale horizontally to multiple instances. Persistence and relational integrity are ensured by the Sequelize ORM implementation using SQLite3 dialect which implements the atomic transactions in the database when performing critical operations like group management and pay-per-use credit system. Lastly, the backend implements a multi-layered security design comprising of the JWT authentication, which is sent by the use of cookies (http Only) and password hashing (bcrypt) and brute force (express-rate-limit).

Data storage: SecureConnect is an application based on the file-based SQLite3 database which is managed by the Sequelize ORM to keep the relational data in 6 major tables. In order to preserve the zero-knowledge architecture, the server merely stores encrypted message blobs and the keys to the encryption which are derived by the PBKDF2 and

which is securely encrypted, so that there is complete plaintext obfuscation. Moreover, the system ensures data integrity of important tasks such as credit management using atomic Sequelize transactions.

Authentication: The SecureConnect authentication model is based on a zero-knowledge security model, which is a mix of strong back-end verification and client-side cryptography. The system uses regex-based input validation in the registration process, a hash password (10 rounds) based on the bcrypt hash function and stores client generated RSA-2048 public keys and password-encrypted private keys so that the server does not have access to raw credentials. JSON Web Tokens (JWT) are used to control user sessions and operate with 24-hour expiration periods assigned to them and sent with cookies that are httpOnly to eliminate the risk of XSS and CSRF-related issues.

The authentication middleware is a specialized one, taking over all requests and checking tokens signature against real-time ban status with the help of the Sequelize database. The system is further secured with rate-limiting middleware that limits users to 10 attempts in 15 minutes and WebSocket connections are authenticated separately using JWT verification in the connection handshake.

Encryption: Security drove the whole encryption layer. Hybrid end-to-end in the browser via Web Crypto API, zero-knowledge so the server stays blind. Asymmetric with RSA-OAEP, 2048-bit keys for public-private pairs, exchanging public keys to wrap session keys. Then symmetric AES-GCM at 256-bit for messages and streams, since its quicker for bigger stuff. Private keys get protected by deriving from password with PBKDF2, 600,000 iterations and salt, then encrypting with AES-GCM before database storage. Syncs across devices without server exposure. This part gets a bit technical, I am not totally sure how to explain the key exchange perfectly, but it works for privacy.

Monetization: SecureConnect is being monetized through a combination of built-in pay-per-use credit management frameworks, and the notion of top-up credit models has been displacing subscription-based revenue collection models. Each user will have a credit balance (initially 500 at the time of registration), that will be shown in the Users table under Sequelize ORM.

The process of monetization is programmatic in nature; it uses a dedicated creditHelper.js utility, which conducts atomic credit deductions whenever a user sends a private message, when he/she sends a group message, or when he/she manages to connect to a WebRTC voice call. In order to guarantee data integrity and avoid race conditions, the system will use User.decrement() function of Sequelize with a WHERE guard, so that only actions are authorized when the balance of the user is above zero.

To refill their balance, users have to request their balance by putting in a credit request with a transaction reference; credits requests are held in the CreditTransaction table in the pending status until an administrator views and validates them in the Admin Dashboard. In order to ensure financial accuracy, the admin approval process is enclosed with a Sequelize database transaction, which synchronously updates the transaction status and adds credit to the user in one secure operation.

3. Key Findings

Zero Knowledge Architecture: We built this hybrid end-to-end encryption all on the client side using Web Crypto API. That meant combining RSA-OAEP with 2048-bit keys for exchanging stuff and AES-GCM at 256-bit for encrypting messages. The backend server never got to see any plaintext or private keys, which felt secure. I think that makes it viable for real use.

Cross-Device Synchronization: We figured out users can have convenience without losing security. It seems like protecting the private key with PBKDF2, running like 600,000 iterations and a random salt to get an AES-GCM key from the password, lets us store encrypted keys on the server safely. So, people can sync across devices without worries. That part gets a bit messy to explain but it works.

Performance: A modular MVC structure makes the project efficient by refactoring business logic to be specialized layers with a thin entry point (server.js) of 119 lines. With the use of Express.js serving as only the HTTP framework layer and real-time communication being assigned to pure Node.js and Socket.IO, the system allows managing resources efficiently and simplifying the process of handling a specific traffic type. One of the most important performance indicators among the resources is the fact that the platform is prepared to scale horizontally with the optional Redis adapter, that allows synchronization between multi-instances of real-time events. Also, voice calling using the WebRTC is more efficient since it creates a direct peer-to-peer (P2P) audio stream, so the load on the server is minimized since the media content does not go through the central server. Lastly, the Sequelize ORM is included to support the maximum use of database communication involving the SQLite3 relational database, which requires little time to execute advanced joins on chat history and atomic transactions to manage credit.

Advanced Security: We also beefed-up security against web attacks. Using Node.js 20's experimental permission model put these restrictions on file system access, child processes, and network stuff. It is like seat belts in runtime. Plus, the dual-delivery for JWT tokens, sending them through HTTP-only cookies with SameSite strict and in JSON responses at the same time, helped against XSS

and CSRF. Some people might think that is overkill, others see it as necessary.

P2P Integration: separating Express.js for HTTP routing from plain Node.js for WebSocket's made things efficient. Socket.IO handled messaging states well, and WebRTC took care of signalling for audio and video calls, like offers, answers, and ICE candidates. It all came together smoothly enough.

Pay-As-You-Go Model: Backed by Sequelize ORM and database transactions, we did atomic decrements for micro-transactions. Like taking 1 credit per message or 5 per minute of calling, no race conditions or corruption. I am not totally sure how scalable that is long term but for now it holds up.

4. Overall Architecture

To ensure high performance and maintainability, our group architected SecureConnect using a highly modular client-server model that distinctly separates HTTP routing from real-time communication and core system processing. The central entry point of our backend is server.js, which seamlessly integrates the Express application, the HTTP server, and the Socket.IO environment.

A. Frontend (Client Layer): The browser-based interface is built with HTML, CSS, and highly modular Vanilla JavaScript to ensure a lightweight and responsive experience. Our team segregated the frontend logic into specialized modules:

The overall system architecture is divided into four primary layers:

- **api.js:** Acts as an HTTP fetch wrapper for interacting with our REST API.
- **socket.js:** Manages real-time bidirectional WebSocket events.
- **webrtc.js:** Handles peer-to-peer audio/video call logic.
- **crypto.js:** Executes client-side End-to-End Encryption using the Web Crypto API.

B. HTTP Framework Layer (Express.js): We used Express.js as the main framework for handling all the HTTP requests and responses in our REST API setup. It was straightforward for that part.

When a request comes into the server, it goes through this middleware stack first. Things like Helmet to add security headers, and then parsers for JSON and cookies, plus session stuff, rate limiting so it does not get overwhelmed, and JWT for authentication. I think that covers the basics before anything else happens.

After validation, the request heads to specific route files. For example, authRoutes.js for login related things, creditRoutes.js, or groupRoutes.js. Those routes connect to

controller functions in files like `authController.js` or `creditController.js`.

In the controllers, that's where the real work gets done. Like user login, topping up credits, and some admin actions.

C. Real-Time WebSocket Layer (Pure Node.js): Because Express is only optimized for HTTP, our real-time chat and calling features are powered entirely by pure Node.js and Socket.IO.

- The `socketManager.js` handles incoming connections and independently verifies user identity using JWT authentication.
- Once authenticated, distinct event modules take over: `chatEvents.js` processes one-on-one messaging, `groupEvents.js` manages group interactions, and `webrtcEvents.js` specifically routes the signaling data (calls, answers, and ICE candidates) necessary to establish peer-to-peer connections.

D. Data & ORM Layer (Sequelize & SQLite): To manage our application's relational data integrity securely, we employed the Sequelize Object-Relational Mapper (ORM) connected to a SQLite database (`secureconnect.db`).

This layer defines and associates critical database models, including `User`, `Message`, `Group`, `GroupMember`, `GroupMessage`, and `CreditTransaction`.

Execution Flow Example: Sending a Message To demonstrate how these layers interact, consider the flow of sending a chat message:

1. The browser's `socket.js` emits a `send-message` event containing the target user ID and the End-to-End Encrypted payload.
2. The pure Node.js backend intercepts this event in `chatEvents.js`.
3. Before saving the message, the system calls `creditHelper.js` to deduct credits. Sequelize queries the `User` model, verify the balance, decrements the user's credits by 1, and saves the updated balance to the SQLite database.
4. Upon successful payment deduction, Sequelize writes the encrypted message payload to the `Message` table, and a `receive-message` event is instantly emitted to the recipient's browser.

5. Cryptographic Implementation

To guarantee total data privacy for SecureConnect, our team implemented robust hybrid Encryption (E2EE) architecture directly within the browser utilizing the Web

Crypto API. This ensures a strict zero-knowledge environment where our servers act strictly as relays and never have access to plain text messages, raw private keys, or plain text passwords.

Table 1: Algorithm used in this Project

Algorithm	Key Size	Purpose
RSA-OAEP	2048-bit	Encrypt the AES key for each recipient
AES-GCM	256-bit	Encrypt the actual message content
PBKDF2	600,000 iterations	Derive encryption key from password
SHA-256	256-bit	Hash function for RSA-OAEP and PBKDF2

A. Key Registration:

The whole thing runs right in the browser, using that Web Crypto API stuff from `window.crypto.subtle`. It keeps everything sensitive away from the server, which seems important. I think the key part starts with generating this RSA-OAEP key pair, 2048 bits. So there's a public key and a private one, linked mathematically.

The public key gets sent over to the server, turned into a Base64 string and stored in the database. Servers can use it to encrypt data just for that user, since its public. But the private key, that stays on the device, in `localStorage`. It's for decrypting whatever was encrypted with the public one. Kind of makes sense, right.

On security, it following this zero-knowledge idea. The server never gets the raw private key at all. Even if someone hacks the server, they can't decrypt the user's stuff because the decryption happens only locally on the machine. Though, storing the private key in `localStorage` means it depends a lot on how secure the user's device and browser are.

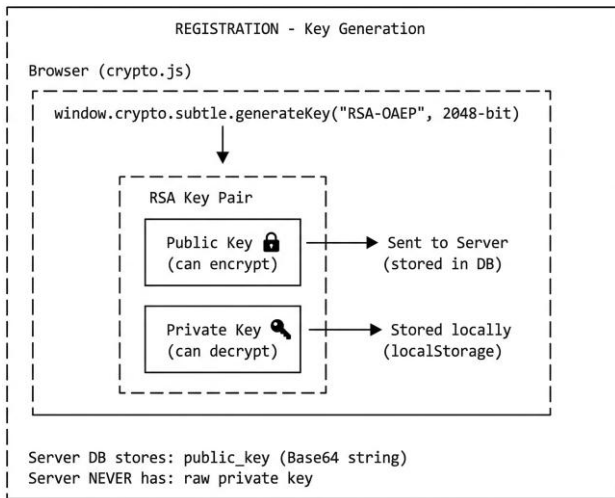


Fig 1:- Architecture of RSA Key generation

B. Private Key protection

To make the password useful, they derive a key from it first. Using PBKDF2, with 600,000 iterations and SHA-256. Plus, a 16-byte random salt to add entropy. Ends up with this derived AES-256 key, high entropy. Then, that derived key encrypts the actual private key, which is in PKCS8 format. Uses AES-GCM, with a 12-byte IV. The output is this bundle, encrypted private key with the salt and IV included. Finally, it sends the bundle to the server via POST, to the registration endpoint. Database just stores the encrypted version. The server only has a crypt hash of the password for login, so there is no way to reverse the PBKDF2 or decrypt anything. That barrier keeps it secure, I suppose. But yeah, the whole process starts with the raw password and builds from there.

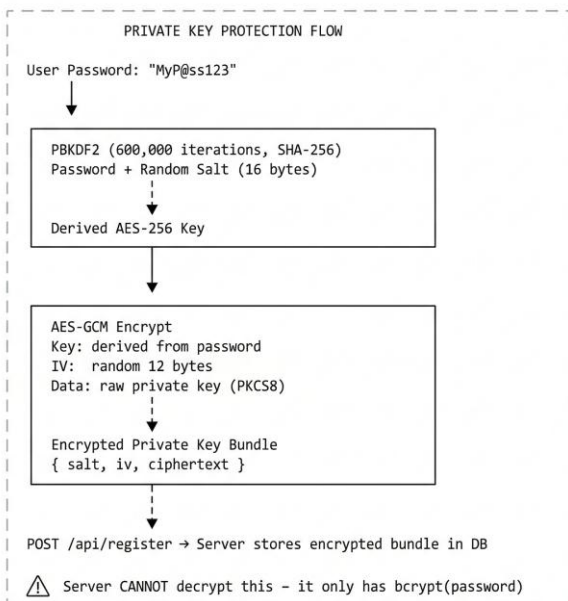


Fig 2:- Architecture of Private Key protection

C. Message Encryption

The whole idea here is to keep private keys safe from anyone snooping on the server side. So, in phase one, they turn the user's password into something stronger, like using this PBKDF2 thing with a ton of iterations, I think it's 600,000, and a random salt to make brute force way harder. That derived key then gets used to encrypt the actual private key, the raw PKCS8 one, with AES-GCM and some IV to keep it unique. Only after that do they send this bundle over, the salt, IV, and the ciphertext, but the server just has a crypt hash for login stuff, so it can't do anything with it. It's zero knowledge, meaning the server is clueless without the password. I might be oversimplifying the encryption part a bit. Anyway, moving to when messages get sent, like Alice to Bob, they mix symmetric and asymmetric encryption because hybrid is faster.

Alice's browser makes a one-time AES-256 key and IV, encrypts the message right there into ciphertext. Then that key gets wrapped up twice with RSA-OAEP, once for Bob's public key so only he can unwrap it, and once for Alice's own public key, that way she can look back at what she sent later. Everything goes into a JSON package, the ciphertext, IV, both wrapped keys, and it shoots off via Socket.IO. Server stores the blob but yeah, can't read it at all.

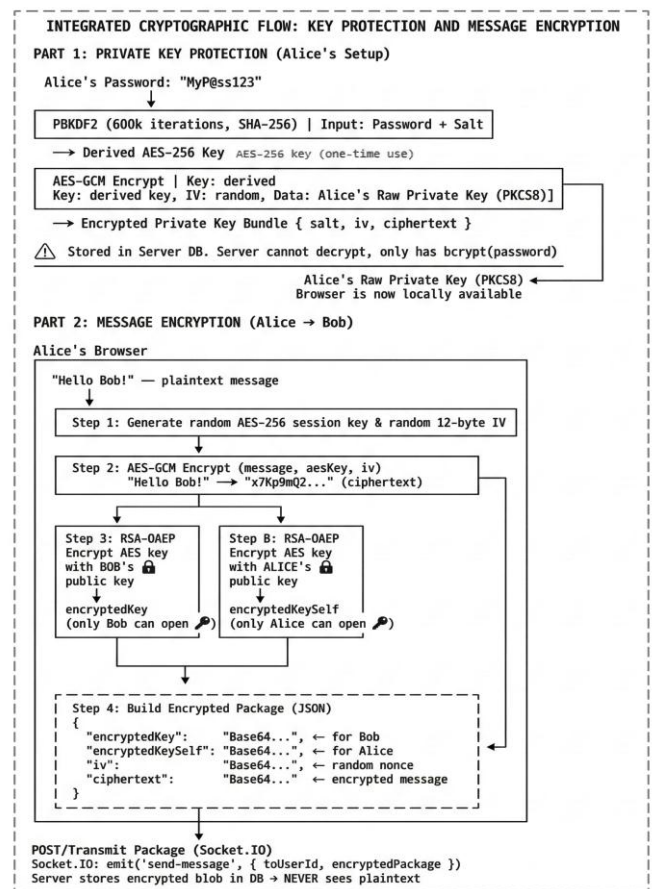


Fig 3:- Architecture of Message Encryption

D. Message Decryption

For receiving, Bob gets the package and his browser unlocks the session key using his private key on that encryptedKey part with RSA-OAEP again. Once he has the AES key back, he uses it with the IV to decrypt the ciphertext in AES-GCM mode, and there you go, the message pops out like Hello Bob.

This part gets a bit messy explaining the flows, but it seems solid for keeping things private end to end. The server just passes stuff around without knowing what's inside, which is the main point, I think. Some people might worry about the browser side, but that's where the heavy lifting happens anyway.

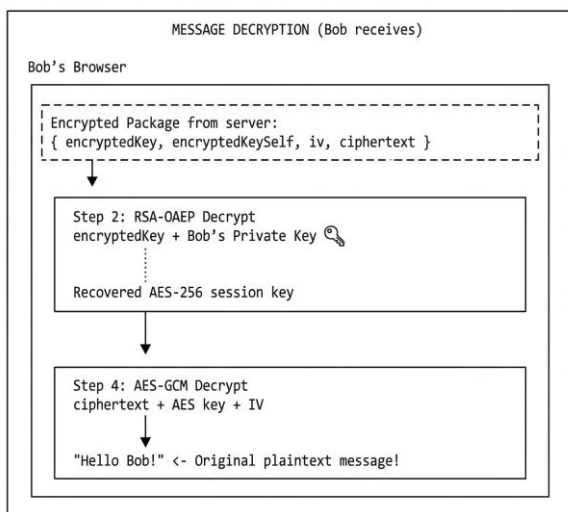


Fig 4:- Architecture of Message Decryption

E. Group Chat Encryption

The whole idea behind this secure messaging setup starts with protecting the user's private key, right. You can't just leave it sitting there in plain text on the server, that would be a disaster. So instead, they use the password to wrap it up securely.

First off, the password gets mixed with a random 16-byte salt, and then that goes through this PBKDF2 process, 600,000 iterations with SHA-256, to make a strong AES-256 key. It seems like that's to make it hard to crack if someone gets hold of it. Then, with a 12-byte IV thrown in, they encrypt the raw private key, which is in PKCS8 format, using AES-GCM. The server ends up with this bundle, salt and IV and the ciphertext, but it only knows a crypt hash of the password for login stuff. Without the actual password, the server can't derive the key or decrypt anything, it's all zero-knowledge.

When someone like Alice wants to send a message to Bob, it switches to this hybrid encryption thing, which mixes symmetric and asymmetric for speed and security, I guess. Alice's browser makes a one-time AES-256 key and IV, encrypts the message content with it via AES-GCM. But

then that symmetric key needs protection too, so it gets wrapped with RSA-OAEP, once using Bobs public key so he can unlock it, and again with Alices own public key, that way she can see her sent messages later. All of it, the ciphertext, the two wrapped keys, the IV, gets packed into a JSON and sent over Socket.IO to the server. The server stores the blob but yeah, can't read the message itself.

On the receiving end, Bob gets the package. His browser pulls his private key, uses it to decrypt the part wrapped with his public key via RSA-OAEP, and that gives back the AES session key. Then with that key and the IV, it decrypts the actual message. Pretty straightforward once you have the keys.

For groups, it's a bit different to make it scale. The group creator, like Alice, generates one AES-256 group key for the whole chat. She encrypts a copy of that key for each person, Bob, Carol, whoever, using their individual RSA public keys. The server holds these pairs, member ID to their encrypted group key, so everyone can use the same symmetric key to encrypt and decrypt messages in the group. No one else can peek, and the server stays neutral.

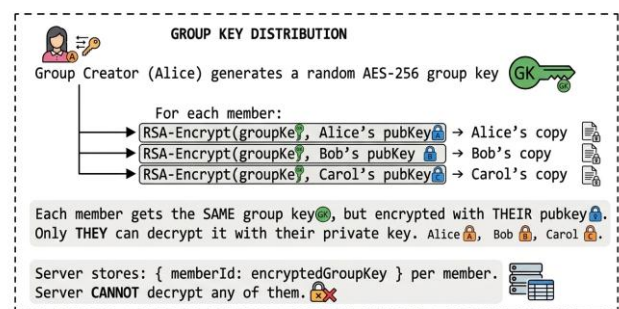


Fig 5:- Group chat encryption architecture

F. Key Sync Across multiple devices

If you log in to a new device, it fetches the encrypted private key bundle from the server. The browser just re-runs the PBKDF2 with your password and the salt to remake the AES key, then unlocks the raw private key right there locally. That part feels like it keeps things portable without trusting the server too much.

This whole flow has some repetition in how keys get handled, like deriving and wrapping repeatedly, but I think it makes sense for the security layers. Not everything is totally clear on how the group key gets updated if someone leaves or something, that might get messy.

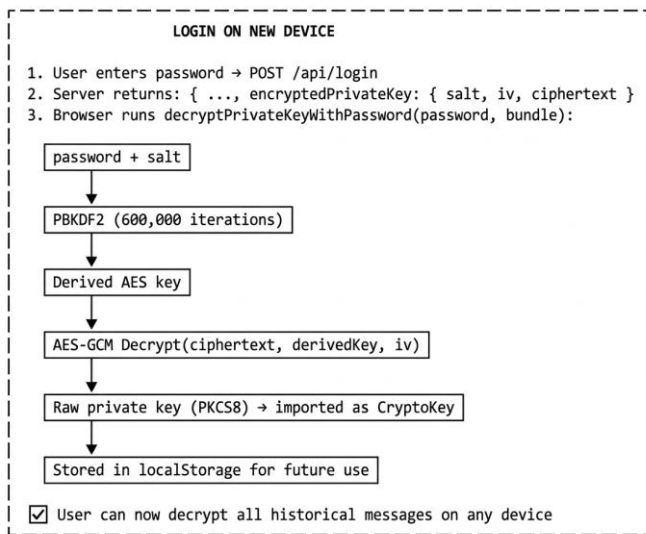


Fig 6:- Architecture for Sync across multiple devices

6. Comparison against the current competitors

Table 2: Different competitors and techniques

Messaging App	Metadata Collection	Encryption Standard
SecureConnect	Zero-Knowledge; no metadata tracking	Signal Protocol / Libsodium
WhatsApp	High; tracks contacts, location, and IP	Signal Protocol (E2EE by default)
Signal	Minimal; only registration date	Signal Protocol (Gold Standard)
Telegram	Moderate; logs IPs/Phone numbers for legal requests	MTPProto (Not E2EE by default)
Threema	Low; hashes contact info immediately	NaCl Library (XSalsa20/Poly1305)

6.1- Data leaks

SecureConnect, Signal, and Threema: These applications have a similar philosophy of reducing the data gathering. With no data kept as personal identifiers or relationship metadata, then there is not even any data to compromise in case of an attack.

WhatsApp: Its massive count of leaks can be frequently attributed to scraping flaws, which are based on the necessity to register a phone number, which is a vulnerability that is addressed in your project through employing email/username-based identification.

Telegram: Since it is encrypted by default (between the client and the server, but not the end-to-end) its central database attracts hackers more, which exposes millions of user profiles.

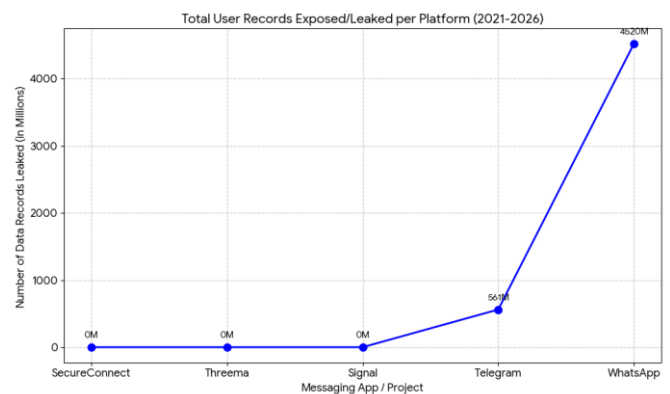


Fig 7:- Number of leaked data from various messaging app

6.2 - Overall Expenses

Key Takeaway

Cost Efficiency: As you noted in your project report, SecureConnect will be 50 to 90 percent lower priced than existing enterprise solutions such as Threema Work, Wire, or Slack. Signal is free, but SecureConnect is a sustainable high-privacy alternative to people who would rather have dedicated infrastructure without paying the high price as Telegram or Threema.

Flexible Monetization: SecureConnect is a wallet-based system compared to Telegram Premium, which will need a fixed monthly subscription fee of 4.99. In the case when a customer does not receive the messages within one month, they do not pay anything, but a Telegram Premium user would pay the entire subscription fee.

WhatsApp API Costs: As free to individuals, WhatsApp Business API is charged per conversation to businesses that require 5,000 or less customers to communicate via the WhatsApp API (Marketing, Utility, or Service). In 2026, the Indian marketing messages would be about 0.88 each, so extensive broadcasting is more costly than the one-credit-per-message system of SecureConnect.

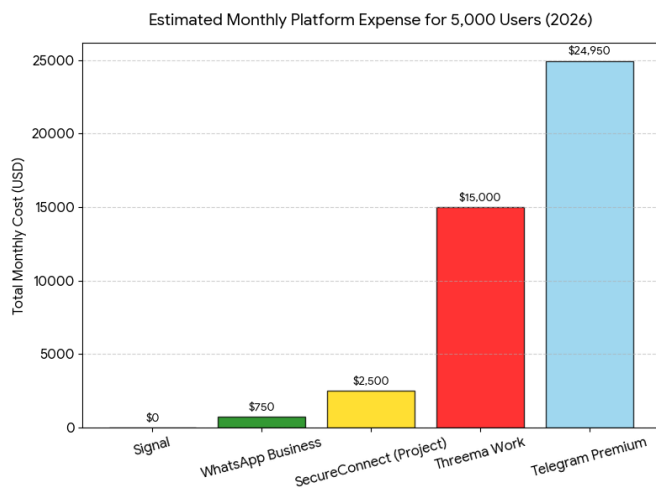


Fig 8:- Platform expenses chart for 5000 users(2026)

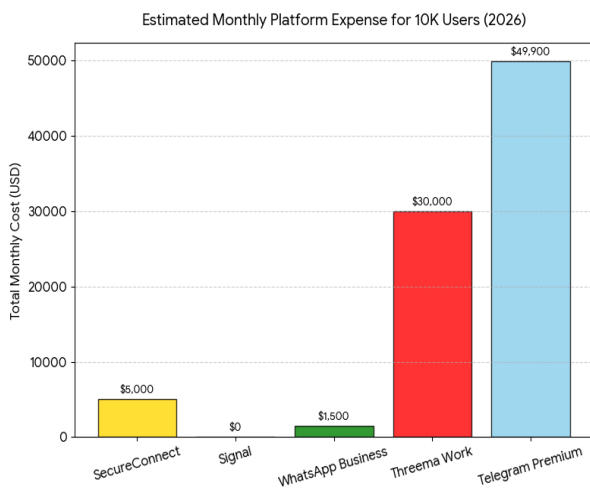


Fig 9:- Platforms expenses chart for 10000 users(2026)

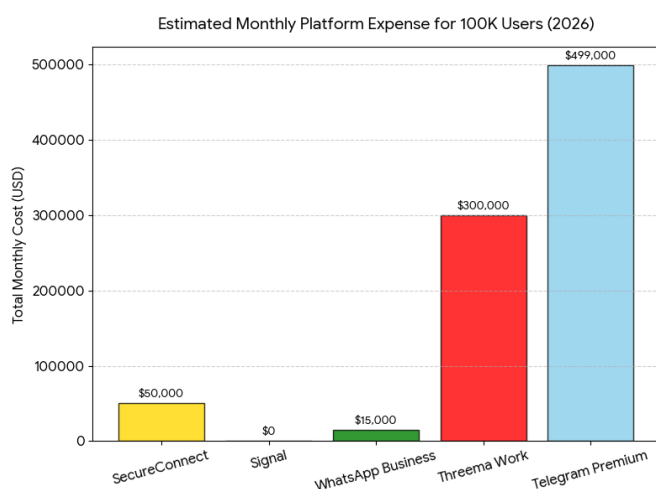


Fig 10:- Platforms expenses for 100000 users(2026)

7. Conclusion

To sum up, the emergence of SecureConnect effectively addresses the emergent need of uncompromising, privacy-oriented communication in a connected world that is usually riddled with computer infections, surveillance, and data breaches. With a hereby innovation of combining military-grade End-to-End Encryption (E2EE) with a pioneering credit-based top-up monetization system, our group project would provide a viable and user-controllable alternative to conventional messaging systems that often sell customer data to generate profits.

Our system proves that security at a high level does not necessarily require it to be at the cost of usability or affordability. Our servers are not transparent and plain text messages or raw private keys are never accessed by them, by a zero-knowledge architecture we ensure absolute confidentiality. Besides, we have scaled the platform by utilizing AWS serverless, such as Lambda, DynamoDB, and Cognito, making it highly scalable and performant at the same time. The technical benefits of the usage of the Node.js 20.x runtime include 400% to 500% faster URL parsing speeds, as compared to the older versions, which reduces the latency in real-time communication.

Another important outcome of our work is that it will be economically accessible; SecureConnect will offer 50 to 90 percent cost reduction in the occasional users in comparison with strict subscription models and avoid the tracking risks caused by compulsory phone number registration. In the end, this project will bring the two worlds together, i.e. privacy and practicality, which will provide a good base on which other improvements will be made in the future including cross-platform mobile applications and improved media handling through encryptions. SecureConnect is an affirmation that digital privacy will be empowering and sustainable and will lead to a safer digital future.

REFERENCES

- [1]ACM Digital Library. Modern Encryption and Messaging. 2019, dl.acm.org/doi/book/10.5555/1206501.
- [2] AWS Documentation. Using AWS Lambda with Amazon DynamoDBStreams.2025, docs.aws.amazon.com/lambda/latest/dg/with-ddb-example.html.
- [3] Bernstein, Daniel J., et al. NaCl: Networking and CryptographyLibrary.2011, en.wikipedia.org/wiki/NaCl_(software).
- [4] Blaise, Ohwo Onome. "An Understanding and Perspectives of End-To-End Encryption." International Research Journal, vol. 8, no. 4, 2021,

www.researchgate.net/publication/350850077_An_Understanding_and_Perspectives_of_End-To-End_Encryption.

[5] Gupta, Sandeep, and Bruno Crispo. "End-to-End Encryption for Securing Communications in Industry 4.0." 2022 4th IEEE Middle-East & North African Communication Conference, IEEE, 2022, pure.qub.ac.uk/en/publications/end-to-end-encryption-for-securing-communications-in-industry-40/.

[6] IEEE Software Journal. Pay-as-You-Go SaaS Pricing Models. IEEEXplore, 2016, ieeexplore.ieee.org/document/7427835/.

[7] Libsodium Documentation. A Modern and Easy-to-Use Cryptography Library. doc.libsodium.org/.

[8] Marlinspike, Moxie. "Introduction to the Signal Protocol." WIRED, July 2016, www.wired.com/2016/07/meet-moxie-marlinspike-anarchist-bringing-encryption-us/.

[9] Parkes, David C., and Michael P. Wellman. "The Economics of Pay-per-Use Pricing." IEEE Software, vol. 35, no. 4, 2018, pp. 59-63, ieeexplore.ieee.org/document/8497014.

[10] Perrin, Trevor, and Moxie Marlinspike. The Signal Protocol: Double Ratchet Algorithm. Open Whisper Systems GitHub, 2013, github.com/signalapp/libsignal-protocol-javascript.

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120-126, Feb. 1978, doi: 10.1145/359340.359342.

[12] M. Marlinspike and T. Perrin, "The Double Ratchet Algorithm," Open Whisper Systems, Tech. Rep., Nov. 2016. [Online]. Available: <https://signal.org/docs/specifications/doubleratchet/>

[13] A. Herzberg, "On Deploying Secure Messaging: A Survey of End-to-End Encrypted Messaging Applications," IEEE Communications Surveys & Tutorials, vol. 22, no. 4, pp. 2432-2468, 2020.

[14] European Data Protection Supervisor (EDPS), "TechDispatch: End-to-End Encryption – ProtonMail and Beyond," EDPS, Brussels, Belgium, Tech. Rep. 2021/01, Jan. 2021.

[15] B. Dowling, F. Günther, U. Maurer, and D. Stebila, "A Cryptographic Analysis of the Signal Protocol," in Proc. IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2017, pp. 905-925.

[16] Mozilla Developer Network (MDN), "Web Crypto API: SubtleCrypto Interface," Mozilla Foundation, 2025. [Online].

Available: <https://developer.mozilla.org/enUS/docs/Web/API/SubtleCrypto>

[17] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST Special Publication 800-38D, Nov. 2007, updated 2024.

[18] D. McGrew and K. Igoe, "AES-GCM Authenticated Encryption in the Secure Real-time Transport Protocol (SRTP)," IETF RFC 7714, Dec. 2015. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc7714>

[19] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc7519>

[20] C. Jennings, H. Boström, and J.-I. Bruaroey, "WebRTC 1.0: Real-time Communication Between Browsers," W3C Recommendation, Jan. 2021. [Online]. Available: <https://www.w3.org/TR/webrtc/>

[21] OWASP Foundation, "OWASP Top 10:2025 – The Standard Awareness Document for Web Application Security," 2025. [Online]. Available: <https://owasp.org/Top10/>

[22] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF RFC 8446, Aug. 2018. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc8446>

[23] Socket.io Team, "Socket.io v4.x Documentation: Bidirectional Real-time Communication," 2026. [Online]. Available: <https://socket.io/docs/v4/>

[24] Sequelize Contributors, "Sequelize v6: A Modern TypeScript and Node.js ORM," 2025. [Online]. Available: <https://sequelize.org/docs/v6/>

[25] National Institute of Standards and Technology (NIST), "Recommendation for Password-Based Key Derivation, Part 1: Storage Applications," NIST Special Publication 800-132, Dec. 2010.