

Ellipticc Drive

End-to-End Encrypted, Post-Quantum Secure Cloud Drive

Ilias ABOUSALIM – ilias@ellipticc.com

November 2025

1. Overview

This whitepaper outlines the security architecture, threat model, and design principles behind Ellipticc—a privacy-first, end-to-end encrypted cloud storage solution with post-quantum cryptography (PQC) for long-term security. Our mission is to provide individuals and organizations with a secure, transparent, and resilient platform for storing and sharing files, ensuring that sensitive data remains private, now and in the future.

Ellipticc guarantees that no data—whether documents, files, or any other sensitive information—is ever stored or accessed by anyone other than the creator and their authorized collaborators. We believe that only the data owner should have control over their information, and that's why we use end-to-end encryption paired with PQC to future-proof security against quantum threats.

In line with this vision, Ellipticc's architecture ensures that no third party, including the platform itself, can access or decrypt user data. We incorporate modern cryptographic standards, including NIST-approved post-quantum algorithms, to protect against the emerging risks posed by quantum computing, ensuring that your data stays secure for decades to come.

At a practical level, Ellipticc combines cutting-edge privacy technologies with an easy-to-use interface to provide users with the ability to store, manage, and share files with confidence. Whether you're an individual safeguarding personal documents or a business storing sensitive client data, Ellipticc offers a solution that puts privacy and security first.

This whitepaper presents the key technical components of Ellipticc, including:

- **Post-Quantum Cryptography:** Using encryption algorithms designed to withstand future quantum computing threats.
- **End-to-End Encryption:** Ensuring only the user and their chosen collaborators can access sensitive data.
- **Secure Data Sharing:** Introducing innovative methods for sharing files securely with trusted parties, without sacrificing privacy.
- **Transparency and Trust:** An open-source frontend and secure, auditable backend.

By incorporating these elements, Ellipticc offers a secure, transparent, and resilient platform where users retain full ownership and control over their data, ensuring their privacy is upheld both today and in the future.

2. Threat Model

Our threat model assumes that adversaries may intercept any data transmitted between the client and server, even over encrypted connections, and that data stored on cloud servers cannot be inherently trusted to remain confidential. In this context, we adopt an "honest but curious" approach to our servers. This means that while our servers faithfully provide services and do not intentionally disrupt access or serve compromised client applications, the data they handle or store could be vulnerable to exposure. To mitigate these risks, particularly for users with high privacy requirements, we implement additional security measures, such as integrity checks for web resources and out-of-band public key verification, ensuring that data integrity is maintained and that cryptographic keys are properly authenticated. While our platform is currently web-only, and native apps could provide extra safeguards against

certain network-based threats, we prioritize the robustness of our web security protocols to deliver a secure experience. Our system also takes proactive steps to protect users from abuse and maintain anonymity, preventing user enumeration and allowing users to block others, remove themselves from shared content, or report misuse. Importantly, we do not collect personally identifiable information during signup, reinforcing our commitment to protecting user privacy from the outset.

1. End-to-End Encryption (E2EE) of All Sensitive Data

- **File & Metadata Protection:** Every document uploaded or created in the cloud is encrypted end-to-end, including metadata (e.g., title, creation time, last modified date), ensuring that only the user or their authorized collaborators can view it.
- **File Sharing:** Shared files remain encrypted, and only recipients with proper decryption keys/link can access them.
- **Post-Quantum Cryptography (PQC):** We use quantum-resistant encryption methods to ensure data security against future quantum computing threats.
- **No Plaintext Storage:** For any file metadata or sensitive information (such as titles or descriptions), no plaintext copies are stored on our servers. This also applies to files shared between users and external services.

2. Protection Against Man-in-the-Middle (MITM) Attacks

- Even if an attacker gains access to the communication channel between the client and server, Ellipticc's encryption ensures that no sensitive information (file contents, names, metadata...) is exposed.
- **Encryption at Rest & In Transit:** All data—whether in storage or during transmission—is encrypted, preventing exposure even if intercepted.

3. Protection Against User Abuse

- We take proactive steps to protect users from harmful behavior, such as unwanted sharing of files or documents.
- **Block & Report Features:** Users can block unwanted contacts, remove themselves from shared files, and report misuse within the platform.
- **Abuse Prevention:** We prevent user enumeration to protect against attempts to gather personal data, and allow full control over who can access shared files.

4. Resistance to Impersonation Attacks

- Our system is designed to prevent impersonation of any party (whether a user or the server itself).
- **Secure Authentication:** Strong user authentication (including multi-factor authentication) ensures that only authorized users can access sensitive data.
- **Session Security:** We implement safeguards against session hijacking and impersonation attempts.

5. Difficult to Phish

- **Account Protection:** We aim to make phishing attacks as difficult as possible. Even if a user's password is compromised, multi-factor authentication (MFA) or hardware tokens will prevent unauthorized access.
- **Key Management:** Encryption keys are never stored on the server, making it more challenging for adversaries to compromise accounts.

6. User-Friendly While Maintaining Security

- **Usability Without Compromise:** While Ellipticc offers significantly better privacy than the majority of cloud storage services, we've worked to make it intuitive and responsive.
- **Easy to Use:** Security features like encryption and MFA are seamlessly integrated into the app's interface, ensuring a smooth and intuitive experience for all users, without sacrificing security.

- **No Security Trade-offs:** We prioritize usability to make sure users don't have to choose between privacy and convenience—enabling them to securely collaborate without switching to less-secure platforms.

2.1. Open-source

Ellipticc has been fully open source since day one, with all our code available for anyone to review, use, and contribute to. We've chosen the MIT License to ensure maximum flexibility and encourage collaboration from developers around the world. By making our platform open source, we not only promote transparency but also empower the community to help shape the future of our project, ensuring it remains secure, private, and aligned with our mission.

3. System Design

3.1. Overview and crypto protocols

End-to-end encryption in our system is designed to securely protect user data and ensure privacy throughout its lifecycle. Each user is assigned a unique public signing key and a separate public encryption key, both of which are paired with corresponding private keys. The signing key is based on Ed25519, and the encryption key uses X25519 for key exchange.

For data encryption, we utilize XChaCha20-Poly1305, which offers strong confidentiality and integrity protection. This encryption method employs XChaCha20 to securely encrypt the data and Poly1305 to authenticate the message(AEAD), ensuring data is both private and unaltered during transmission.

To secure key exchange, we incorporate Kyber768 (ML-KEM768), a post-quantum Key Encapsulation Mechanism (KEM). This ensures the confidentiality of key exchanges even in the face of quantum computing advances. Additionally, Dilithium2 (ML-DSA65), a post-quantum digital signature algorithm (PQC), is used for generating secure signatures, further strengthening authentication and integrity.

Our cryptographic stack uses reliable, open-source libraries such as [@noble/ed25519](#) for Ed25519 signing, [@noble/curves/ed25519.js](#) for X25519 key exchange, [@noble/post-quantum/ml-kem.js](#) for Kyber768 KEM, [@noble/post-quantum/ml-dsa.js](#) for Dilithium2 PQC signatures, [@noble/ciphers/chacha](#) for XChaCha20-Poly1305 encryption, [hash-wasm](#) for hashing functions like argon2id and sha512, and [@scure/bip39](#) for BIP39 mnemonic generation.

To maintain maximum security, private keys must be safeguarded at all times to prevent unauthorized access or tampering. The next section covers how we ensure private key protection.

3.2 Account Creation, Login, and Private Key Protection

Ellipticc's account creation and login system is engineered to:

1. **Safeguard private keys at all times**
2. **Never transmit passwords or key material** outside the user's browser
3. **Resist brute-force and offline dictionary attacks** via strong cryptographic hardening

We use the **OPAQUE asymmetric PAKE exclusively for authentication**, not for key wrapping. After successful authentication, a **client-side Master Key (MK)** derived from the user's password is used to encrypt all cryptographic keypairs. This ensures **zero-knowledge**: the server never sees passwords, private keys, or the MK.

Account Creation Flow

1. **User Input** The user (e.g., Alice) enters:
 - Email address
 - Strong password (enforced: uppercase, lowercase, number, special char)
2. **OPAQUE Registration (Client - Server)**
 - Client and server perform **OPAQUE-2K** registration using [@noble/opaque](#).

- Server generates and stores a **per-user OPRF secret key**.
- Client computes a **password-blinded OPRF output**.
- Result: Server stores an **opaque record** (encrypted export key + OPRF data). **No password, hash, or verifier is ever stored.**

3. Client-Side Key Generation Using @noble/curves and @noble/post-quantum:

```

signingKeypair = Ed25519.keyPair()

encryptionKeypair = X25519.keyPair()

pqKemKeypair = ML_KEM_768.keyPair()

pqSigKeypair = ML_DSA_65.keyPair()

```

4. Master Key Derivation (MK)

Server returns a random 32-byte master_salt (generated once, stored in DB). Client computes:

```

MK = Argon2id(
    password = user_password,
    salt = master_salt,
    params = { m: 64*1024, t: 3, p: 2 } // 64 MiB, 3 iterations, 2 lanes
)

```

5. Encrypt Keypairs with MK

All private keys are serialized and encrypted in-browser:

```

envelope = {
    ed25519_priv, x25519_priv,
    ml_kem_priv, ml_dsa_priv
}

{ ciphertext, nonce } = XChaCha20-Poly1305.Encrypt(MK, envelope)

```

6. Upload to Server

Client sends:

- opaque_record (from OPAQUE)
- master_salt (public)
- encrypted_key_envelope (Base64)
- key_envelope_nonce (Base64)
- All **public keys** (unencrypted)
- Email (for recovery/contact)

Server stores all values. **No plaintext keys or passwords ever touch the backend.**

Login Flow

1. **User enters email + password**
2. **OPAQUE Authentication**
 - Client downloads opaque_record and master_salt
 - Performs OPAQUE login with server
 - On success: **authenticated session** (no keys yet)
3. **Re-derive Master Key (MK)**

```

MK = Argon2id(password, master_salt)

```

4. Download & Decrypt Key Envelope

- Fetch **encrypted_key_envelope** + **nonce**
- Decrypt locally:

```
keypairs = XChaCha20-Poly1305.Decrypt(MK, ciphertext, nonce)
```

- Load private keys into memory
5. **Receive Session JWT**
- Server issues short-lived JWT (e.g., 15 min)
 - Used for file operations, sharing, etc.

This is **clean, auditable, standards-compliant**, and **resilient** — the correct way to do password-based **E2EE** cloud storage.

3.2 Crypto Wallet Login

Ellipticc integrates native authentication through Ethereum-compatible wallets such as MetaMask and Brave Wallet. This mechanism leverages decentralized identities and builds upon the **Sign-In With Ethereum (SIWE)** standard (EIP-4361) to provide secure, passwordless access to user accounts.

Authentication Flow

1. **Wallet Ownership Proof** When a user connects an Ethereum wallet (public address **eth_addr**), the server generates a **random nonce** n and sends it to the client. The user signs a **challenge message** containing the nonce using their wallet:

```
signature = sign("Ellipticc login nonce: " || n, eth_privKey)
```

The client sends **eth_addr**, **n**, and **signature** to the server. The server verifies the signature using the public key derived from **eth_addr**. This proves live control of the private key and prevents replay or impersonation attacks.

2. **Master Key Derivation (Client-Side Only)**

After successful ownership verification, the client derives the Account Master Key (AMK) locally using a static, deterministic message signed by the wallet:

```
static_msg = "Ellipticc master key derivation for " || eth_addr  
wallet_sig = sign(static_msg, eth_privKey) // computed in-browser, never sent  
master_salt = <32-byte random value from server, stored in DB>  
AMK = Argon2id(password = wallet_sig, salt = master_salt, params = {m=64MiB, t=3, p=2})
```

- The `wallet_sig` is **never transmitted or stored**.
- The `master_salt` is generated once per account and stored **unencrypted** in the backend database.
- The resulting AMK is used to encrypt/decrypt the user's Ed25519/X25519 + PQ keypairs (stored as an encrypted rows on the server).

- 3) **Session Establishment**

The client decrypts its keypairs using the derived AMK and receives a short-lived JWT for API access.

Security Caveat

This is **not the most secure authentication path in Ellipticc**. While convenient and fully passwordless, it relies on **key derivation from a wallet signature + server-stored salt**. An attacker who compromises **both**:

- The backend database (gaining master_salt), **and**
- The user's browser environment (extracting wallet_sig via XSS, malware, or session theft) ...can recompute the AMK and decrypt all user data.

Stronger alternatives include:

- **OPAQUE password login** (offline brute-force resistant)
- **Hardware-bound recovery key**
- **Passkey / WebAuthn** (phishing-resistant, no shared secrets)

Fallback for Limited Wallets

For wallets without programmatic signing support outside of EIP-4361 flows, wallet-based signup is **disabled**. Users are redirected to standard **OPAQUE password registration** to maintain security guarantees.

3.4 Two-Factor Authentication (2FA)

Ellipticc supports optional two-factor authentication (2FA) as an additional security layer during account login. This mechanism is based on the open standard Time-Based One-Time Password (TOTP) protocol, enabling compatibility with widely used authenticator applications such as Google Authenticator, Authy, and Duo Mobile.

Setup Phase

When a user enables 2FA, the browser locally generates a new TOTP secret using the [otplib](#) library. This secret, a Base32-encoded random key, acts as the shared secret between the user's authenticator app and Ellipticc's backend servers.

The client displays this secret as a QR code for the user to scan with an authenticator app, thereby provisioning the app with the shared key. The app begins generating 6-digit one-time codes every 30 seconds, following the TOTP standard ([RFC 6238](#)).

To confirm successful setup, the user must enter one valid OTP code. The client then transmits the TOTP secret to the server over a secure HTTPS connection once the code is validated.

Storage and Encryption

Upon receipt, the server encrypts the TOTP secret using symmetric authenticated encryption provided by the [@noble/ciphers](#) library. Specifically, the implementation uses the **XChaCha20-Poly1305** AEAD construction to provide confidentiality and integrity guarantees.

The encryption key used for this operation is securely derived and stored within the server's secret management infrastructure. The resulting ciphertext and nonce are stored in the database as part of the user's account record.

This design ensures that no TOTP secret is stored in plaintext at rest. Even if the database were compromised, the attacker would be unable to recover the underlying key material without access to the server's encryption key.

Verification Phase

During subsequent logins, once the user successfully authenticates with their primary credential (password or wallet signature), the server requests a one-time code from the user's authenticator app.

To verify the code, the backend retrieves the user's encrypted TOTP secret from the database and decrypts it in memory using the same ChaCha20-Poly1305 key. The server then computes the expected OTP value for the current time window using the standard TOTP algorithm:

$$\text{TOTP} = \text{Truncate}(\text{HMAC-SHA1}(\text{secret}, \text{time_step}))$$

If the computed OTP matches the user-provided code (within the acceptable time drift window), authentication succeeds and the login session is established.

After verification, the decrypted secret is immediately cleared from memory to minimize exposure.

3.5 Account Recovery and Password Changes

Elliptic employs a **double-wrapped master key architecture** to enable secure, user-controlled account recovery without compromising zero-knowledge guarantees. The user's **Account Master Key (AMK)**—a 32-byte symmetric key used to encrypt/decrypt all cryptographic key material (Ed25519, X25519, ML-KEM, ML-DSA)—is **never stored in plaintext** on the server. Instead, it is encrypted under **two independent paths**:

1. **Password-Derived Secret (PDS)** – for standard login
2. **Recovery Key (RK)** – for emergency recovery via mnemonic

This design ensures that **even if one path is lost (e.g., forgotten password), the other remains sufficient to regain access**, while preventing the server from ever learning or deriving the AMK.

Password-Derived Path (Standard Login)

1. Client retrieves `account_salt` and `encrypted_AMK_password` from server.
2. Derives:

```
PDS = Argon2id(password, account_salt, {m=64MiB, t=3, p=2})
```

3. Decrypts:

```
AMK = XChaCha20-Poly1305-Decrypt(PDS, encrypted_AMK_password, nonce)
```

4. AMK is cached in **sessionStorage** for the session (never persisted long-term).

Recovery-Key Path (Password Forgotten)

When the user enables **account recovery**, the following occurs **client-side**:

```
// 1. Generate Recovery Key (RK)
RK = crypto.getRandomValues(32)

// 2. Derive RKEK from mnemonic
RKEK = SHA256(SHA256(mnemonic))

// 3. Encrypt RK → encryptedRecoveryKey
{ encryptedRecoveryKey, recoveryKeyNonce } = XChaCha20-Poly1305-Encrypt(RK, RKEK)

// 4. Encrypt AMK → encryptedMasterKey
{ encryptedMasterKey, masterKeyNonce } = XChaCha20-Poly1305-Encrypt(AMK, RK)
```

The server stores:

- **encryptedRecoveryKey + recoveryKeyNonce**
- **encryptedMasterKey + masterKeyNonce**
- **account_salt** (for password path)

User exports and securely backs up the 12-word mnemonic offline.

Recovery Flow

If the user forgets their password:

1. **Verify account ownership** via email passcode (rate-limited, one-time code).
2. User enters **12-word mnemonic** in the recovery interface.

3. Client computes:

```
RKEK = SHA256(SHA256(mnemonic))
RK = XChaCha20-Poly1305-Decrypt(encryptedRecoveryKey, RKEK, recoveryKeyNonce)
AMK = XChaCha20-Poly1305-Decrypt(encryptedMasterKey, RK, masterKeyNonce)
```

4. With AMK recovered, client:

- Decrypts all keypairs
- Prompts user to set a **new password**
- Re-encrypts AMK under new **PDS'**:

```
PDS' = Argon2id(new_password, account_salt)
encrypted_AMK_password' = XChaCha20-Poly1305-Encrypt(AMK, PDS')
```

- Uploads new **encrypted_AMK_password'** to server

No data loss. Full forward secrecy preserved.

Password Change (Without Recovery)

When changing password **while logged in**:

1. AMK is already in memory.
2. Derive new PDS' from new password.
3. Re-encrypt:

```
encrypted_AMK_password' = Encrypt(AMK, PDS')
```

4. Upload new blob. RK path remains unchanged.

Caveats & Best Practices

- **Mnemonic must be stored offline** (paper, metal backup, hardware wallet).
- **Never enter mnemonic on untrusted devices.**
- **Email passcode is rate-limited and expires in 10 minutes.**
- **Recovery disables after 3 failed attempts** (prevents online guessing).

This design resists database breaches, supports password resets, and ensures no user ever loses access to their encrypted data—even if they forget their password.

4. Building a filesystem

In order to support user filesystems in Ellipticc, each document contains a unique 64-character hexadecimal identifier generated via `crypto.randomBytes(32).toString("hex")` and a parent pointer **dp** (also a 64-character hex ID, or **null** for root-level items); the entire filesystem is constructed client-side by decrypting documents with their folder keys and linking them via these parent pointers, ensuring that no structural metadata—such as hierarchy, titles, or paths—is ever visible to the server. This design maintains absolute end-to-end privacy while enabling efficient, scalable sharing and unsharing through folder-level key rotation, with all collaboration models optimized for large numbers of documents and users.

4.1 Scalable sharing and unsharing

To share a file or folder, Ellipticc generates a random **shareId** and decrypts the item's **Content Encryption Key (CEK)** client-side, embedding the raw CEK directly in the URL fragment: `https://drive.ellipticc.com/<shareId>#raw_CEK`. The recipient opens the link, fetches the encrypted document metadata from the backend using only the **shareId**, then

decrypts it locally using the CEK from the fragment — **no private keys, passwords, or server involvement in decryption** are required. The fragment (**#raw_CEK**) is never sent to the server per HTTP specification, ensuring full security and zero trust. To unshare, the owner simply deletes the **shareId** from the backend; any subsequent access results in a “link expired or revoked” error — no key rotation, re-encryption, or complex management needed, enabling instant, scalable revocation.

4.2 Sharing a document dr (recursive filesystem)

Similar to our simple sharing case above, Alice now wants to share a folder **dr** (the root of a tree with hundreds or thousands of documents) with Bob. In this case, Alice simply encrypts dr’s Content Encryption Key (CEK) with Bob’s public key and embeds it in the link fragment: https://drive.ellipticc.com/<shareId>#raw_CEK. Bob opens the link, decrypts dr using the CEK, then recursively decrypts all child documents using their own CEKs (derived or stored within the parent) — **no server involvement, no key exchange per file**. The entire subtree is accessible in **O(1)** sharing operations, with unsharing just as simple: delete the shareId, and the whole tree is instantly revoked.

4.3 Adding a password-gate to a file

To add password protection to a shared link, the owner optionally sets a strong password; the **CEK** is then encrypted directly with a key derived from the password via **HMAC-SHA256**, and the **resulting hashed KDF output** (used for verification) is stored on the backend alongside the **shareId**. The final link remains <https://drive.ellipticc.com/<shareId>> — **no password or CEK appears in the fragment**. When a recipient visits the link, the backend blocks all access — including metadata — until the correct password is entered and verified against the stored hash. Only upon successful verification is the **password-encrypted CEK** released to the client, which then derives the decryption key locally to access the item and its subtree. This ensures **zero server knowledge** of the password or plaintext CEK, while enforcing access control before any data exposure.

4.4 Expiring Access

To enable time-limited sharing, the owner specifies an expiry date when creating the link; this timestamp is stored on the backend alongside the **shareId**. When a recipient attempts to access the share, the server compares the current time to the stored expiry date: if the link has expired, access is denied with a “link expired” message — no metadata or CEK is revealed. This simple server-side check ensures automatic, tamper-proof revocation at the exact deadline, with no client changes or key management required.

4.5 Email invitation link for files

Users can share any file or folder with anyone, even if the recipient is not an Ellipticc user, by sending an **email invitation link**. The share link — https://drive.ellipticc.com/<shareId>#raw_CEK — is first sent to Ellipticc’s backend mail server, which then forwards it to the recipient’s email address. **Note:** because the backend handles email delivery, it **transitively sees the full link including the raw CEK in the fragment** during sending. While this data is **never logged or stored**, the system technically has the capability to do so — a trade-off for convenience in non-registered user sharing (we’ll work on a solution for this later on). For maximum privacy, users should manually copy and send links via encrypted channels when sensitive CEKs are involved.

5. Conclusion

Ellipticc is built for one simple reason: **your data should belong only to you**. From the moment you upload a file to the second you share a folder, everything—names, contents, structure—stays fully encrypted and private. No server, no employee, no third party can ever see it. With post-quantum security, instant sharing, password-protected links, and email invites that work for anyone, we’ve made strong privacy feel effortless. This isn’t just a product—it’s the start of a new kind of internet, where **you stay in control**, today and decades from now.

6. Contact

Please reach out to ilias@ellipticc.com with any questions, feedback, or collaboration ideas!