NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Segurança de Sistemas de Computadores

2025/2026

Implementation of a Secure Encrypted Block Storage Service Using a Client–Server TCP/IP Solution

---

Autores:

72908, Francisco Rodrigues dos Santos

73433, Teresa Domingos

Professor:

Henrique Domingos

Mês 10, 2025

Índice

Introduction

This project implements a Secure Encrypted Block Storage System designed to transform the initial insecure implementation into a robust architecture that guarantees confidentiality, integrity, and authenticity of stored data. The system follows a client–server model in which the server is untrusted and acts only as a storage backend. All cryptographic operations, key management, and verification are performed exclusively by the client. The solution supports encrypted file storage, retrieval, keyword-based search, and block-level deduplication while ensuring that the server never gains access to any plaintext data or secret keys.

System Architecture

The system is organized in two main components: the client and the server.
The client is responsible for all security-critical operations, including encryption, authentication, keyword indexing, and key management. It maintains a persistent local index (client_<id>_ser) that maps each filename to its corresponding list of block identifiers.

The server, on the other hand, is an untrusted storage entity that only handles encrypted data. It stores two persistent structures on disk: the blockstorage/ directory, which contains encrypted file blocks, and the Secure_metadata.ser file, which holds the encrypted searchable metadata. The server never has access to plaintext information or cryptographic keys, ensuring a strong separation between functionality and trust.

Cryptographic Configuration and Design

The cryptographic operations are dynamically defined by an external configuration file (cryptoconfig.txt), which specifies the algorithm and key size to be used. This design enables flexible runtime selection of cryptographic primitives without altering the source code.

The system supports three encryption modes:

- AES-GCM and ChaCha20-Poly1305: both are authenticated encryption with associated data (AEAD) algorithms that provide confidentiality, integrity, and authenticity through a single primitive. A random IV or nonce (12 bytes) is generated for each block, and an authentication tag (typically 16 bytes) is automatically verified during decryption.

- AES-CBC with PKCS5Padding: this mode provides confidentiality only, so a separate HMAC-SHA256 tag is computed over the IV and ciphertext to ensure integrity and authenticity. The final transmitted block contains the IV, ciphertext, and HMAC tag, following an Encrypt-then-MAC pattern.

The decryption process verifies the authenticity of every block before releasing plaintext. In AEAD modes, the tag verification is implicit in the decryption operation, whereas in AES-CBC mode, the client explicitly recalculates and compares the HMAC value. Any authentication failure aborts file reconstruction and prevents corrupted or tampered data from being used.

Key Management

Key management is handled entirely on the client side by the CryptoManager class. When the client starts, it attempts to load its dedicated Java Cryptographic Extension Keystore (JCEKS), stored as KeyStore/<clientID>.jceks and protected by the user's password. If the keystore does not exist, new cryptographic keys are securely generated using Java's SecureRandom and then stored in the keystore.

The keystore holds three keys:

- Encryption key (sKey): used for block encryption and decryption.
- Authentication key (hmac): used only in AES-CBC mode for HMAC verification.
- Searchable encryption key (seKey): used to generate deterministic tokens for the secure metadata index.

When the encryption mode specified in cryptoconfig.txt is changed, the system checks whether a corresponding encryption key already exists in the client's keystore. If no key for that mode is found, new keys specific to the selected algorithm are generated and added to the existing keystore. Previously stored keys are never deleted, allowing the keystore to hold multiple encryption keys for different configurations.

This design ensures that secret keys are never stored in plaintext and that only the user who knows the password can access them. The password protects the keystore, which guarantees confidentiality and persistence of the cryptographic material between sessions.

File Storage and Deduplication

When a user uploads a file using the PUT command, the client divides it into fixed-size blocks of 4096 bytes. For each block, the system computes its SHA-256 hash, which serves as the unique block identifier (block ID). This approach enables deduplication, as identical plaintext blocks will always produce the same hash and therefore be stored only once on the server.

Each block is encrypted according to the selected cryptographic mode. The client then sends the command STORE_BLOCK to the server along with the block ID and the encrypted block data. The server checks if a file with the same block ID already exists in the blockstorage/ directory. If it does, the server skips writing the new data, achieving efficient deduplication while maintaining complete data opacity.

The client also updates its local index file (client_<id>_ser), maintaining the ordered list of block identifiers associated with each filename. This index is serialized to disk to ensure persistence between sessions.

Searchable Encryption

The system supports privacy-preserving keyword search through a deterministic Searchable Encryption mechanism. For every keyword provided during the PUT operation, the client computes a token using the function HMAC-SHA256(keyword, seKey). The resulting token is then Base64-encoded and used as an index key. The associated file identifier (fileId) is also encrypted using AES-GCM with the main encryption key (sKey) to maintain its confidentiality.

The mapping between each keyword token and the encrypted file identifier is sent to the server using the STORE_METADATA command. The server stores these mappings in the serialized metadata file Secure_metadata.ser.

When a user performs a SEARCH operation, the client regenerates the deterministic token for the requested keyword and sends it to the server. The server looks up the token and returns the list of encrypted file identifiers that match. The client then decrypts them locally to obtain the original filenames, allowing keyword-based search without revealing the actual keywords or file identities to the server.

File Retrieval and Integrity Verification

During a GET operation, the client retrieves each encrypted block from the server using its block ID and decrypts it through the CryptoManager. The decryption process includes automatic integrity verification, either through AEAD tag validation or HMAC comparison, depending on the encryption mode. If verification fails for any block, the process stops and the partially reconstructed file is deleted to prevent the use of corrupted data. Successfully

verified blocks are combined to reconstruct the original file, which is saved locally as retrieved_<filename>.

Security Guarantees and Persistence

The implemented system provides end-to-end confidentiality, integrity, and authenticity. Confidentiality is achieved because all stored data and metadata remain encrypted, and only the client holds the necessary cryptographic keys. Integrity and authenticity are ensured through authenticated encryption or explicit HMAC verification, making it impossible for an attacker or the untrusted server to modify data undetectably.

Deduplication improves storage efficiency without compromising security, while searchable encryption allows privacy-preserving keyword queries. Persistence is achieved through disk-based storage on both the client and the server: the server maintains encrypted blocks and metadata, and the client retains its keystore and local file index. Together, these mechanisms provide a robust, secure, and fully functional encrypted block storage system that satisfies all the project requirements.

Conclusion

In conclusion, the final implementation successfully transforms the original insecure reference system into a secure and privacy-preserving storage platform. It enforces confidentiality through encryption, integrity and authenticity through authentication tags or HMACs, and functionality through searchable encryption and deduplication. The design ensures that the untrusted server cannot access, modify, or infer any user data. All keys remain securely protected within the client's password-encrypted keystore, guaranteeing that only the rightful user can decrypt or search their stored files. The system is therefore compliant with the intended security objectives and demonstrates the successful application of modern cryptographic principles in a practical client–server environment.