
# Work Assignment 1 –A Searchable Secure Encrypted Block Storage Service

Name: Nicole Arquissandas

Student number: 75026

# 1-Introduction

This project implements a secure block-based storage system written in Java, composed of a client and a server.
The objective is to ensure that files can be:

- securely stored as encrypted blocks on an untrusted server,
- searched through keywords without revealing their content,
- and later retrieved or verified for integrity.

The system fulfills the following security goals:

1. **Confidentiality:** only the client can encrypt and decrypt data.
2. **Integrity:** any modification of a block or metadata is detectable.
3. **Keyword privacy:** keywords are never sent in plaintext.
4. **Modularity:** cryptographic algorithms are configurable at runtime via cryptoconfig.txt, without recompiling code.

# 2. System Architecture

The system consists of two main components:

## 2.1 Client

The client handles:

- File splitting and block encryption.
- Metadata management (local indexes).
- Generation of search tokens (HMAC of keywords).
- Reconstruction and integrity checking of files.

Main classes:

- ClientCore — orchestrates all client operations.
- CryptoFactory and CryptoSuite — manage cryptographic configuration and algorithm selection.
- CLTest — command-line program that tests all functionalities (PUT, GET, SEARCH, LIST, CHECKINTEGRITY).

## 2.2 Server

The BlockStorageServer stores only encrypted blocks and keyword metadata.
It has no knowledge of:

- the original filenames,
- the content of the files,
- or the plaintext keywords.

It simply maintains:

- blockstorage- directory containing encrypted block files.
  - Each block is stored as a binary file named <fileId>_block_<index>.
- metadata.ser - a serialized map of { token → Set<fileIds> }.

The server **never performs decryption**, it simply receives encrypted data and returns it on demand, therefore functions as a **secure but untrusted repository**.

# 3. Data Flow: From File to Encrypted Blocks

What happens internally when the client executes:

## 3.1 PUT Command

Steps:

1. The client generates a random **fileId** (UUID).
   Example: 44e199d2f5c749b193fe154f5dfccb98
2. The file is read in blocks of 4096 bytes
3. For each block:
   - The Associated Authenticated Data (AAD) is computed as:

     ```
     FileId:blockIndex
     ```

     Example: 44e199d2f5c749b193fe154f5dfccb98:0
   - The block is encrypted using the chosen suite:

     ```
     encryptedBlock = SUITE.encrypt(plaintext, aad)
     ```
   - The result is a concatenation of iv || ciphertext || tag (for AES/GCM) or the equivalent for the chosen algorithm.
   - The first block also sends the HMAC-sha256 tokens of the associated keywords.
4. The encrypted block is sent to the server:

```
STORE_BLOCK
blockId = <fileId>_block_<index>
```

The server stores the raw binary file in its blockstorage directory.

5. After all blocks are uploaded, the client saves metadata locally:
    - client_index.ser → filename → list of blockIds
    - client_fileids.ser → filename → fileId
    - client_keywords.ser → filename → keywords

## 3.2 Keyword Token Generation

Each keyword is converted to a search token using an HMAC-sha256 with the KWKEY:

```
Mac mac = Mac.getInstance("HmacSHA256");
mac.init(new SecretKeySpec(KWKEY, "HmacSHA256"));
token = Base64.getEncoder().encodeToString(mac.doFinal(keyword.getBytes()));
```

- The same keyword always produces the same token.
- The server never sees plaintext keywords.
- Different clients with different keys produce unrelated tokens.
- The KWKEY is different than the key used for encryption
- These tokens are sent only once (for the first block) and used by the server for the inverted index.

**Example:**

Keyword: dog

Token: Z4q7dZp8C8yoC4G7SYZwoA==

## 3.3 SEARCH Command

When a user runs:

```
java CLTest SEARCH dog
```

The client:

1. Computes the same token for “dog”.
2. Sends command "SEARCH" + token.
3. The server looks up metadata[token] → returns all matching file IDs.
4. The client uses its local map fileIds to translate file IDs back to filenames:

```
44e199d2f5c749b193fe154f5dfccb98 → dog.txt
```

## 3.4 GET Command

```
java CLTest GET dog.txt retrieved/
```

Or

```
java CLTest GET <dog.txt keyword> retrieved/
```

The client:

1. Retrieves all block IDs associated with that file from fileIndex.
2. Sends "GET_BLOCK" for each block to the server.
3. For each block:
    - Receives the encrypted blob.
    - Decrypts using the suite:
    - Writes the plaintext to retrieved/dog.txt.

The AAD ensures that even if a block is swapped or reordered, decryption fails.

## 3.5 CHECKINTEGRITY Command

Verifies:

1. **Block integrity:**
   Every block is re-downloaded and re-decrypted. If a GCM tag ,HMAC or POLY1305 tag fails, the client reports an integrity failure.

2. **Keyword integrity:**
   The client re-generates each keyword token and re-queries the server.
   If its fileId is not in the result, integrity fails.

Example output:

```
Blocks OK (integrity verified).

Keywords OK (2).

CHECKINTEGRITY: PASS for dog.txt
```

# 4. Cryptographic Design

All cryptography is centralized in the **CryptoFactory** and **CryptoSuite** classes.

## 4.1 CryptoFactory

- Reads cryptoconfig.txt to determine which algorithm suite to use.
- Supports:
  - AES_GCM
  - AES_CBC_HMAC
  - CHACHA20_POLY1305
- Creates an instance of the corresponding suite:

```
if ("AES_GCM".equalsIgnoreCase(cfg.alg))
   return new AesGcmSuite(dataKey);
else if ("AES_CBC_HMAC".equalsIgnoreCase(cfg.alg))
   return new AesCbcHmacSuite(dataKey, macKey);
else if ("CHACHA20_POLY1305".equalsIgnoreCase(cfg.alg))
   return new ChaCha20PolySuite(dataKey);
```

This ensures that changing algorithms only requires editing cryptoconfig.txt, not recompiling.

## 4.2 Key Management

All client-side keys are generated and stored persistently in client_keys.properties:

```
DATAKEY = <base64>
KWKEY = <base64>
MACKEY = <base64>   (only for AES_CBC_HMAC)
```

These keys are loaded at startup by:

KEYS = loadOrCreateKeys();

- DATAKEY: used for encrypting file blocks.
- KWKEY: used to compute keyword HMAC-SHA256 tokens.
- MACKEY: used to compute block HMACs (AES_CBC_HMAC only).

If the file doesn't exist, the client automatically generates new random keys using SecureRandom.

# 5. Security Properties

| Property | Achieved By | Notes |
|---|---|---|
| **Confidentiality** | AES-GCM / AES-CBC / ChaCha20 encryption | Server cannot decrypt blocks |
| **Integrity** | GCM tag ,HMAC-SHA256 or POLY1305 tag | Detects any block modification |
| **Authentication** | Implicit through MAC verification | Ensures only client with keys can validate |
| **Keyword Privacy** | HMAC-SHA256 tokens | Server never learns keywords |
| **Configurability** | cryptoconfig.txt | No recompilation needed |
| **Local Metadata Protection** | Separate local .ser files | Prevents mixing between clients |