Work Assignment 1 : Frequency Evaluation [version 1.1]:

Note: This version has small clarifications on the version 1.0 (published in 9/Oct) in section 7

Title: A Searchable Secure Encrypted Block Storage Service

Objective. The objective of this assignment is to design and implement a client–server system using TCP sockets (as a possible primary solution), where the server provides a persistent, encrypted block storage service for clients to store files as encrypted blocks. The system must guarantee the confidentiality, integrity, and authenticity of stored file blocks while supporting basic file operations and metadata-based search functionality, providing a solution for searchable encryption of blockstorage contents when clients want to search files by using keywords used as metadata. This assignment covers network programming, operating systems, file system management, and applied cryptography, giving students experience in implementing secure client–server communication and secure storage, with practical guarantees for confidentiality, integrity, and authenticity, including a solution for searchable encryption.

2. Requirements. The system to be designed, implemented, experimentally evaluated, and demonstrated is based on a client–server architecture. Communication between the client and the server is supported by TCP/IP. The server must provide a secure, encrypted block storage service, allowing clients to perform remote operations, including:

1. Storing encrypted file blocks
2. Retrieving and reconstructing files from remotely stored blocks
3. Listing files associated with remotely stored encrypted blocks
4. Searching for files associated with remotely stored searchable encrypted blocks

The client and server requirements are as follows:

2.1 Server-Side Requirements

a) Implement a block storage service where each file sent by a client is split into fixed-size blocks, which are always stored and maintained in encrypted form on the server.
b) Ensure that stored blocks are encrypted and persistently maintained on disk, guaranteeing confidentiality.
c) Implement mechanisms to ensure the integrity and authenticity of each stored encrypted block, preventing unauthorized modification or tampering that can be detected by clients during remote operations.
d) Maintain metadata for each file, including keywords associated with it, to support server-side search through a searchable encryption solution.
e) Support the following TCP-based operations that clients can perform:
    a. **STORE_BLOCK:** Receive and persist encrypted blocks.
    b. **GET_BLOCK:** Retrieve encrypted blocks for file reconstruction.
    c. **LIST_BLOCKS:** List stored files corresponding to encrypted blocks.
    d. **SEARCH:** Search for files by keywords, using metadata associated with the stored blocks.
f) Persistently store both blocks and metadata in encrypted form on disk, ensuring data survives server restarts.

2.2 Client-Side Requirements

a) The client maintains a local index mapping filenames to their corresponding block IDs, which are sent and stored in encrypted form on the server.
b) Encrypt and authenticate blocks with integrity guarantees before sending them to the server.
c) Support the following operations:
   a. **PUT:** Split local files into blocks, encrypt them, and store them on the server along with associated metadata.
   b. **GET:** Retrieve encrypted blocks from the server, verify their integrity and authenticity, decrypt them, and reconstruct the original file.
   c. **LIST:** Display a list of files stored in the client's local index.
   d. **SEARCH:** Query the server for files using metadata keywords.
d) The client persistently stores its local index on disk to maintain state across restarts.

3. Reference of Usage Scenarios

Storing a File (PUT)	• The client has a local file, *report.pdf*, and wants to store it securely on the server. • The client splits the file into blocks, encrypts each block, generates a MAC or digital signature for integrity and authenticity, and sends the blocks to the server along with metadata keywords, e.g., *finance*, *Q3*. • The server stores the encrypted blocks persistently on disk and records the metadata. • The client updates its local index with the mapping: *report.pdf → [block0_id, block1_id, …]*.
Retrieving a File (GET)	• The client requests *report.pdf* from the server. • The server sends the encrypted blocks. • The client verifies the integrity and authenticity of each block, decrypts them, and reconstructs the original file locally. • If any block has been tampered with, the client detects it and aborts reconstruction.
Listing Stored Files (LIST)	• The client lists all files it has stored using its local index. • Example output: **Stored files:** o report.pdf o presentation.pptx o notes.txt
Ensuring Security Guarantees	• **Confidentiality:** Encrypted blocks cannot be understood if intercepted. • **Integrity:** Any modification of stored blocks is detected upon retrieval. • **Authenticity:** The client can confirm that blocks were created by a legitimate client. • **Searchable Encryption:** It will be possible to search files stored in encrypted bocks by keywords, using these keywords submitted as metadata, ex: o The client searches for files containing the keyword *finance*. o The server examines the metadata associated with stored blocks and returns files matching the keyword. o Example server response: *report.pdf, budget.xlsx*.

4. A client test to demonstrate your implementation.

Independently of, or in addition to, any client implementation in your project, you must develop a specific program (or script) for a test client that will allow you to quickly test your implementation. This test client (or script) must comply with the following reference specifications (running with the indicated arguments to show the provided functionality)

cltest PUT <path/dir/file> <keywords>	Upload a file from the local client directory <path/dir> to the block storage server as encrypted blocks. Include metadata keywords for possible search. Files can be binary or text of any size.
cltest LIST	List all files currently stored as encrypted blocks in the block storage.
cltest SEARCH <keywords>	Search the block storage for encrypted blocks associated with the given keywords and identify the corresponding file(s).
cltest GET <file> <path/dir>	Retrieve and reconstruct the file <file> stored as encrypted blocks in the block server, saving it to the local client directory <path/dir>.
cltest GET <keywords> <path/dir>	Retrieve and reconstruct one or more files from encrypted blocks in the block server that match the provided keywords, saving the file(s) to the local client directory <path/dir>.
cltest GET CHECKINTEGRITY <path/dir/file>	Validate the integrity of encrypted blocks stored in the blockstorage server as well as, keywords, of a previously file submitted to the blockstorage server with the PUT command

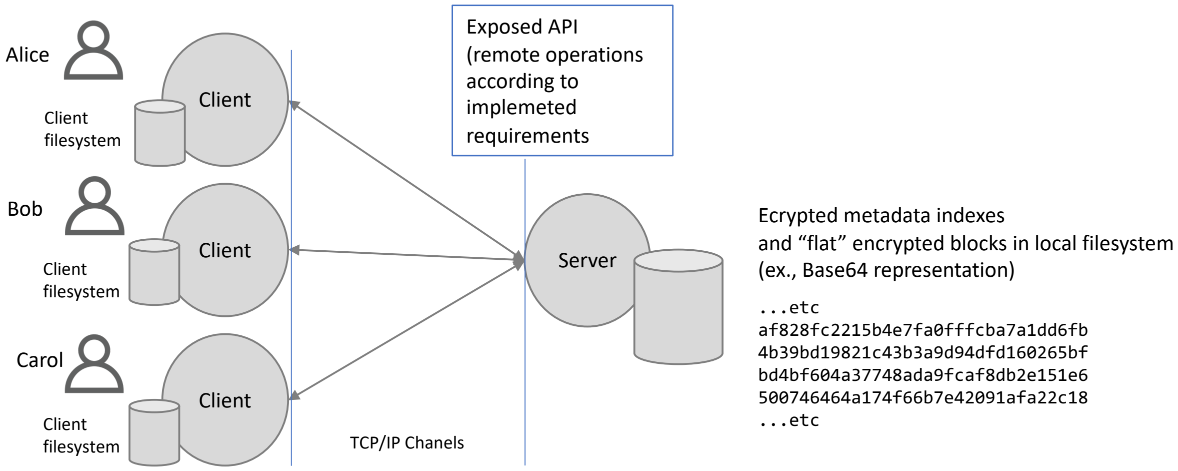
5. Opaque Content in the Filesystem of the Encrypted Block Storage Server

Observing the contents of a directory where the server stores encrypted blocks and/or associated encrypted metadata reveals no information about the actual data, even to a system administrator accessing the server filesystem.

For example, when examining a directory in the server's local filesystem that contains file chunks (with block sizes determined by the client), the listed content may appear as follows (represented in Base64):

af828fc2215b4e7fa0fffcba7a1dd6fb
4b39bd19821c43b3a9d94dfd160265bf
bd4bf604a37748ada9fcaf8db2e151e6
500746464a174f66b7e42091afa22c18
... etc

The following figure shows the system model assumptions


Alice
Client
Client filesystem
Bob
Client
Client filesystem
Carol
Client
Client filesystem
Exposed API (remote operations according to implemeted requirements
Server
TCP/IP Chanels
Ecrypted metadata indexes and "flat" encrypted blocks in local filesystem (ex., Base64 representation)

```
...etc
af828fc2215b4e7fa0fffcba7a1dd6fb
4b39bd19821c43b3a9d94dfd160265bf
bd4bf604a37748ada9fcaf8db2e151e6
500746464a174f66b7e42091afa22c18
...etc
```

6. Security Guarantees

The solution must be able to implement security guarantees against the following adversary model conditions:

- Adversaries in communication channels between clients and the blockstorge server, performin attacks to break OSI X.800 securiry properties (not considering Denial f Server or Distributed Denial of Server attacks)
- Adversaries that have access to the server to leak file contents (indexed and submitted by clients) trying to break confidentiality of client stored files for possible leakage of stored files (that we will characterize here as a "Honest-But-Curious" adversary (including HbC System administrator of the blockstorage server machine), trying to learn about the contents or from the blockserver execution context.

Given the adversary model assumptions, the following security guarantees must be provided:

- **Confidentiality:** The server stores blocks in encrypted form as file chunks with generated filenames. Only the client can decrypt these blocks. On the server side, it must not be possible to obtain the content, and neither the content nor the original filenames can be inferred from the names of the stored encrypted blocks.
- **Integrity:** Any modification to stored blocks must be detectable by the client upon retrieval or checked whenever the client needs to "attest" the integrity expected in remote content related to stored files (as encrypted blocks)
- **Authenticity:** The client must be able to verify that retrieved blocks originated from a legitimate source (i.e., the client that initially stored the blocks) and have not been tampered with.
- **Searchable Encryption Support:** A key security goal is enabling Searchable Encryption (SE), a cryptographic technique that allows a client to store encrypted data on an untrusted server while still being able to search over it without decrypting it on the server. This combines confidentiality with query capability. Your solution must allow clients to search the encrypted repository directly by keywords, without first retrieving and reconstructing the stored blocks and associated files.
- Normally, if data is encrypted for confidentiality (e.g., with AES), the server cannot understand or search it. Searching would require downloading and decrypting all data, which is inefficient and defeats the purpose of remote encrypted storage.
- The searchable encryption solution must allow the server to perform searches without learning the content of the data or the keywords (or at least minimize what it learns). Stored blocks remain encrypted on the server, protecting confidentiality even against "honest-but-curious" administrators with high privileges.
- To support searchable encryption, the client generates a secure index for keywords or generates search tokens (trapdoors) that the server can use to check if a document matches a keyword without learning the keyword itself. The secure index must be encrypted, ideally using a deterministic encryption scheme.

Suggested Approach:

1. The client encrypts files and creates an encrypted keyword index, for example:
   keyword: [encrypted_doc_id1, encrypted_doc_id2, ...]
2. The client sends the encrypted files and index to the server.
3. When the client wants to search for a keyword, e.g., "healthcare":
   - The client generates a search token (trapdoor) for "healthcare."
   - The server uses the trapdoor to find matching encrypted documents (i.e., stored encrypted blocks) without knowing the keyword.

- The server returns the matching encrypted blocks to the client, who decrypts them.

4. The client can attest in any moment the integrity of maintained encrypted blocks or searchable metadata (with or without retrieving the related files)

Benefits of Searchable Encryption in this system:

- **Confidentiality:** The server cannot see plaintext data nor infer meaningful information by accessing stored data.
- **Query privacy:** The server cannot determine which keyword is being searched in any practical computational time.
- **Access pattern leakage (optional):** Some SE schemes may leak which encrypted documents match searches. Advanced schemes can hide even this, though it is not required for this project.

7. Implementation Guidelines:

- The solution must use standard cryptographic primitives (e.g., AES for encryption, HMAC or digital signatures for integrity/authenticity, AES-GCM, or ChaCha20-Poly1305).
- The client should be able to select any of these options via a configuration file (e.g., cryptoconfig.txt). Your implementation must be designed to run with the specified options **without requiring recompilation** of the client or server.
- Examples for entries in the cryptoconfig.txt file include:

AES/GCM KEYSIZE: 256 bits	AES/CBC/PKCS5Padding KEYSIZE: 256 bits HMAC-SHA256 MACKEYSIZE: 256 bits	CHACHA20-Poly1305 KEYSIZE: 256
Only uses AES/GCM mode (as known, it is already an AEAD-based encryption, no need for Padding. Modes or additopnal MACs)	Uses AES/CBC/ PKCS5Padding and Block Integrity and Authenticity with HMAC-SHA156	Only uses CHACHA20 (as known, it is already an AEAD-based encryption, no need for Padding. Modes or addental MACs)

- You can manage keys on the client side using local keystore files or key files, where keys may be stored in Base64 or hexadecimal format.
- Note that only the client needs to maintain the keys (encryption keys or HMAC keys).
- To improve security when storing keys in a client file, you may encrypt the file using a password, which derives a protection key to securely access the configured keys inside the file.
- You can also address the use of keystores (for storing keys) and the use of keytool to generate keys or any additional required secrets to be managed by keystores (ex., as provided in Java)
- If you want to consider, it is also possible to avoid the use of stored keys or keystores by using mechanisms based on Password-Based Encryption or generated keys on the fly in the client, using secure key derivation functions from strong passwords, compatible with the required cryptographic configurations above.

8. Technology

a) The solution can use TCP sockets for communication between the client and server.
b) Use disk-based persistent storage on the server filesystem to maintain encrypted blocks and metadata (on the machine where the block storage server is running).
c) The solution requires a local client index stored on the client filesystem.
d) Provide a simple command-line interface on the client to perform the required operations: PUT, GET, LIST, and SEARCH.

e) The work assignment can be designed and developed in Java, Kotlin, or Go. Students are free to choose the programming language that best suits their convenience.
 For Java, you can base your solution on the provided initial implementation, which is not secure (and must be analyzed). See the initial material in **PA1-Material** (Java Implementation). For Python, an insecure initial implementation is also provided (which must be analyzed).

Initial Code (Java and Python) Provided to Start Development

- **Server:** Stores blocks on disk and maintains metadata for each file (including keywords). Supports STORE_BLOCK, GET_BLOCK, LIST_BLOCKS, and SEARCH based on metadata keywords.
- **Client:** Maintains a local index mapping filenames to block IDs. Can PUT, GET, LIST, and SEARCH files using keywords.

Initial materials provided and features initially implemented

a) **Persistent Block Storage:** Blocks are stored on the server side as files in the blocks/ directory.
b) **Client Index:** Maps filenames to a list of block IDs, stored locally.
c) **File Reconstruction:** GET reconstructs files from blocks.
d) **List Files:** LIST displays files in the local index.
e) **Search by Keyword:**
    a. Keywords are stored as metadata on the server (for the first block of each file), indexed in metadata.ser.
    b. SEARCH returns blocks whose metadata contains the keyword.
f) **Persistent Client Index:** Saved to client_index.ser across sessions.

As observed, all information on the server side is stored in plaintext. Therefore, the initial code is **not a secure solution**, as we intend.

9. Evaluation criteria

Your solution must be secure and implement the required security properties. For this assignment, the minimum specific security mechanisms required are: symmetric cryptography—either stream ciphers with AEAD protection or block ciphers using AEAD modes, or block ciphers combined with MACs.

Ref. Spec.	Criteria	Evaluation
Provided material	Use the initial code, analysis, compilation and run/test he initially provided functionality, investigating the limitations compared with the required solution	Total: 2
Section 1 and Section 3	Server-Side Requirements: implementation and experimental validation of requirements, including support for: PUT, GET, LIST (as required)	Total: 3 (1 by each operation
Section 2 and Section 3	Client-Side Requirements: implementation and experimental validation of requirements, including support for: PUT, GET, LIST (as required)	Total: 3 1 by each operation
Section 1 and Section 3	Server-Side Requirements: implementation and experimental validation of requirements, including support for SEARCH (as required)	Total: 2
Section 2 and Section 3	**Client-Side Requirements:** implementation and experimental validation of requirements, including support for SEARCH (as required)	Total: 2
Section 4	Opaque content in the filesystem of the encrypted block storage server is guaranteed	Total: 2

Section 5	Resistance against the adversary model conditions and implemented Security Guarantees	Total: 3
Section 6	Support for deduplication of stored encrypted blocks and support for different clients using the block storage server concurrently	Total: 2
TOTAL:		20

You must address the mandatory requirements (following the evaluation grid and indicative criteria), but you may also propose and implement additional improvements for demonstration, which can be considered for bonus evaluation. In this case, up to 2 bonus points may be awarded for each well-considered and effectively demonstrated idea. Here we give you some ideas:

- For example, you could implement your solution using a client/REST service architecture. Students are free to use any programming tools, technologies, or components to support such a solution. This approach also opens the possibility of deploying the Block Storage Service in a cloud environment, which can be an interesting and valuable extension.

  - Using a client/REST service approach (e.g., a REST-based web app implementing the Secure Block Storage Server) exposes you to relevant challenges and applications, such as:

    - Encrypted cloud storage with search functionality (e.g., Google Drive- or Dropbox-like systems).
    - Secure email search.
    - Healthcare or financial databases, where data is sensitive and must remain secure and private, yet searchable even in encrypted form.

- You could design and implement an authentication system for clients authorized to use the blockstorage service. You can create manual registrations of users and user credentials (example: users and passwords), with all necessary information provided maintained with the same security guarantees as you must have for the provided operations).

- You could consider implementing the blockserver as a fully docker-based isolated system exposing he necessary operations for remote clients to access

- You could consider creating a solution using a model for blockstorage replication resisting to the possible failure of one-only centralized blockstorage server

- You could address a partitioned solution, where blocks are stored in more than one server (as a fragmented blockstorage solution), somewhat similar to the approach to a simple encrypted RAID-like approach implemented at software level

10. Deliverables, Delivery Instructions and Important Dates

See information in Project1-Delivery-Instructions, and additional questions related to project implementation