ENSIAS

École Nationale Supérieure d'Informatique et d'Analyse des Systèmes - Rabat

File Integrity Verification using SHA-256 and AWS S3 Integration

Cryptographic Hashing: Theory and Practice with SHA-256

Student :
WIAME YOUSFI

Professor:
Miss. SOUAD SADKI

Contents

1 Introduction

1.1 Context and Motivation

In today's digital landscape, ensuring data integrity has become paramount for cybersecurity. Organizations and individuals need mechanisms to verify that their data has not been tampered with, corrupted, or modified without authorization. Cryptographic hashing provides a robust solution to this challenge by generating unique digital fingerprints for data.

This lab work explores the practical application of SHA-256, one of the most widely adopted cryptographic hash functions, to verify file integrity. Through hands-on implementation, we demonstrate key concepts including hash generation, the avalanche effect, and integration with cloud storage services.

1.2 Objectives

The primary objectives of this laboratory exercise are:

1. **Understand Cryptographic Hashing**: Grasp the fundamental principles and properties of cryptographic hash functions

2. **Implement SHA-256**: Develop a practical Java application for generating SHA-256 hashes

3. **Demonstrate the Avalanche Effect**: Show how minimal input changes result in drastically different hash values

4. **Detect File Modifications**: Use hash comparison to identify data tampering

5. **Integrate Cloud Storage**: Implement AWS S3 integration for secure hash and file storage

6. **Apply Security Best Practices**: Understand proper implementation of hash-based integrity verification

1.3 Lab Overview

This lab is structured in multiple phases:

- **Phase 1**: Development environment setup and AWS configuration

- **Phase 2**: Implementation of hash generation functions

- **Phase 3**: File modification and hash comparison

- **Phase 4**: Cloud storage integration with AWS S3

- **Phase 5**: Analysis and security implications

2 Theoretical Background

2.1 What is Cryptographic Hashing?

A cryptographic hash function is a mathematical algorithm that transforms an input (message) of arbitrary length into a fixed-size output called a hash value or digest. This transformation is deterministic, meaning the same input always produces the same output.

2.1.1 Mathematical Representation

A hash function H can be formally represented as:

$$H : \{0,1\}^* \to \{0,1\}^n \tag{1}$$

Where:

- $\{0,1\}^*$ represents the set of all possible binary strings of any length (the input space)
- $\{0,1\}^n$ represents the set of all binary strings of fixed length n (the output space)
- For SHA-256: $n = 256$ bits (32 bytes)

2.2 Essential Properties of Secure Hash Functions

For a hash function to be cryptographically secure, it must satisfy several critical properties:

2.2.1 Determinism

The same input must always produce the same hash value:

$$\forall x_1, x_2 \in \{0,1\}^* : x_1 = x_2 \implies H(x_1) = H(x_2) \tag{2}$$

2.2.2 Pre-image Resistance (One-wayness)

Given a hash value h, it should be computationally infeasible to find any input x such that:

$$H(x) = h \tag{3}$$

This property ensures that hash functions are irreversible in practice.

2.2.3 Second Pre-image Resistance

Given an input x_1 and its hash $H(x_1)$, it should be computationally infeasible to find a different input x_2 where:

$$x_1 \neq x_2 \text{ and } H(x_1) = H(x_2) \tag{4}$$

2.2.4 Collision Resistance

It should be computationally infeasible to find any two different inputs x_1 and x_2 such that:

$$H(x_1) = H(x_2) \text{ where } x_1 \neq x_2 \tag{5}$$

2.2.5 Avalanche Effect

A small change in the input (even a single bit) should produce a significantly different hash value. Ideally, changing one bit should change approximately 50% of the output bits.

2.3 SHA-256 Algorithm

SHA-256 (Secure Hash Algorithm 256-bit) is part of the SHA-2 family designed by the National Security Agency (NSA) and published by NIST in 2001.

2.3.1 Technical Specifications

Property	Value
Output Size	256 bits (32 bytes)
Internal State Size	256 bits
Block Size	512 bits
Number of Rounds	64
Security Level	128 bits
Word Size	32 bits

Table 1: SHA-256 Technical Specifications

2.3.2 Algorithm Steps

The SHA-256 algorithm processes input data through the following stages:

1. **Padding**: The input message is padded to ensure its length is congruent to 448 modulo 512
2. **Length Appending**: A 64-bit representation of the original message length is appended
3. **Initialize Hash Values**: Eight 32-bit hash values are initialized
4. **Process Message Blocks**: The message is divided into 512-bit blocks
5. **Compression Function**: Each block undergoes 64 rounds of operations
6. **Final Hash**: The final hash value is produced by concatenating the hash values

2.4 Comparison of Hash Algorithms

Algorithm	Output Size	Block Size	Security	Status
MD5	128 bits	512 bits	Broken	Deprecated
SHA-1	160 bits	512 bits	Broken	Deprecated
SHA-256	256 bits	512 bits	Strong	Recommended
SHA-512	512 bits	1024 bits	Strong	Recommended
SHA-3-256	256 bits	Variable	Strong	Recommended

Table 2: Comparison of Common Hash Algorithms

3 Laboratory Setup and Configuration

3.1 Development Environment

3.1.1 Required Software

The following software components are required for this laboratory:

- **Java Development Kit (JDK)**: Version 17 or higher
- **Apache Maven**: Build automation and dependency management
- **AWS CLI**: Command-line interface for AWS services
- **IDE**: IntelliJ IDEA (recommended) or Eclipse
- **AWS Account**: Active Amazon Web Services account

3.1.2 Project Structure

The project follows standard Maven directory structure:

```
file-hashing-lab/
 pom.xml
 src/
    main/
        java/
           com/
               hashinglab/
                   FileHashingLab.java
        resources/
            original_file.txt
 file_hash.txt
 modified_file.txt
 modified_file_hash.txt
 README.md
```

3.2 Maven Configuration

The `pom.xml` file defines project dependencies and build configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.hashinglab</groupId>
    <artifactId>file-hashing-lab</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <!-- AWS SDK for S3 -->
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>s3</artifactId>
            <version>2.20.26</version>
        </dependency>

        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>auth</artifactId>
            <version>2.20.26</version>
        </dependency>
    </dependencies>
</project>
```

3.3 AWS Configuration

3.3.1 AWS CLI Setup

Configure AWS CLI with your credentials:

```
aws configure
```

This command prompts for:

- AWS Access Key ID
- AWS Secret Access Key
- Default region name (e.g., `us-east-1`)
- Default output format (e.g., `json`)

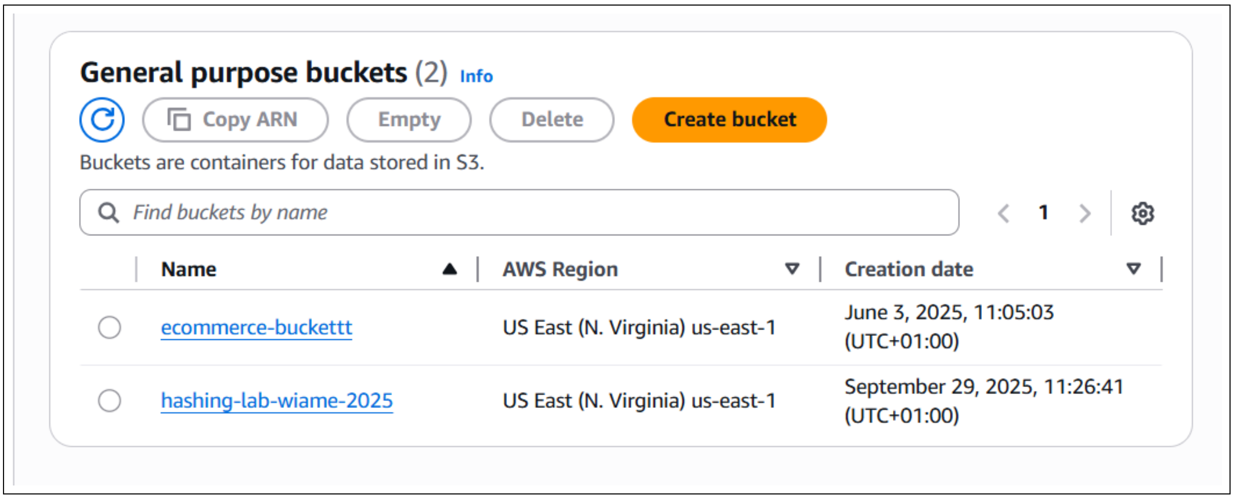
@default@default

Figure 1: AWS CLI Configuration

3.3.2 S3 Bucket Creation

Create a dedicated S3 bucket for the laboratory:

```
aws s3 mb s3://hashing-lab-wiame-2025 --region us-east-1
```



General purpose buckets (2)

Buckets are containers for data stored in S3.

Name	AWS Region	Creation date
○ ecommerce-buckettt	US East (N. Virginia) us-east-1	June 3, 2025, 11:05:03 (UTC+01:00)
○ hashing-lab-wiame-2025	US East (N. Virginia) us-east-1	September 29, 2025, 11:26:41 (UTC+01:00)

Figure 2: S3 Bucket Created Successfully

4 Implementation

4.1 Core Hash Generation Function

The core functionality implements SHA-256 hash generation using Java's `MessageDigest` class:

```java
public static String generateSHA256Hash(String filePath)
    throws Exception {
    // Initialize SHA-256 digest
    MessageDigest digest = MessageDigest.getInstance("SHA-256");

    // Read file contents
    byte[] fileBytes = Files.readAllBytes(Paths.get(filePath));

    // Calculate hash
    byte[] hashBytes = digest.digest(fileBytes);

    // Convert to hexadecimal representation
    StringBuilder hexString = new StringBuilder();
    for (byte b : hashBytes) {
        String hex = Integer.toHexString(0xff & b);
        if (hex.length() == 1) hexString.append('0');
        hexString.append(hex);
    }

    return hexString.toString();
}
```

Key Implementation Details:

- Uses `MessageDigest.getInstance("SHA-256")` to obtain SHA-256 algorithm
- Reads entire file into byte array using `Files.readAllBytes()`

- Applies hash function using `digest.digest()`
- Converts resulting bytes to hexadecimal string representation

4.2 Hash Storage Function

This function saves hash values with metadata to a text file:

```java
public static void saveHashToFile(String hash, String outputPath)
    throws IOException {
    try (BufferedWriter writer =
         new BufferedWriter(new FileWriter(outputPath))) {
        writer.write("SHA-256 Hash:\n");
        writer.write(hash);
        writer.write("\n\nGeneration date: " +
                     java.time.LocalDateTime.now());
    }
}
```

4.3 AWS S3 Upload Function

Integration with AWS S3 for secure cloud storage:

```java
public static void uploadToS3(String bucketName,
                               String keyName,
                               String filePath) {
    Region region = Region.US_EAST_1;
    S3Client s3 = S3Client.builder()
            .region(region)
            .credentialsProvider(ProfileCredentialsProvider.create())
            .build();

    try {
        PutObjectRequest putOb = PutObjectRequest.builder()
                .bucket(bucketName)
                .key(keyName)
                .build();

        s3.putObject(putOb, RequestBody.fromFile(new File(filePath)));
        System.out.println(" File uploaded successfully to S3: " +
                           keyName);
    } catch (Exception e) {
        System.err.println("Error during S3 upload: " +
                           e.getMessage());
    } finally {
        s3.close();
    }
}
```

4.4 Hash Comparison Function

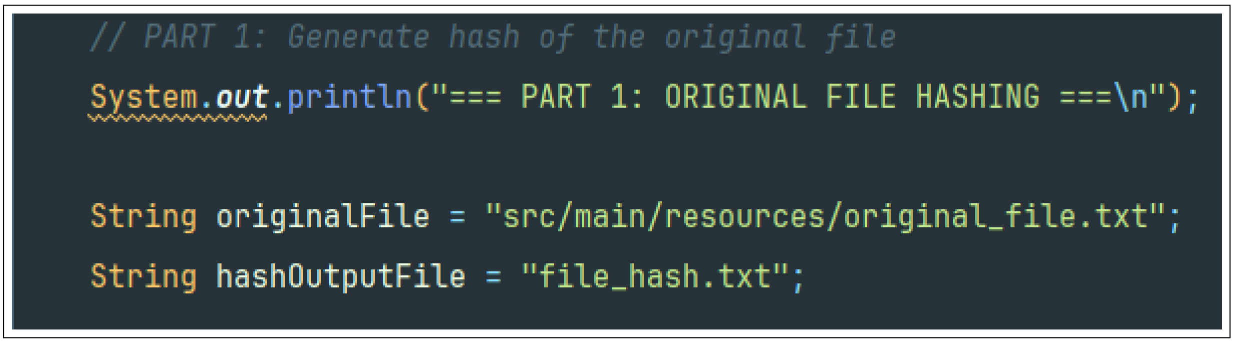
This function performs detailed comparison between two hash values:

```java
public static void compareHashes(String hash1, String hash2,
                                 String label1, String label2) {
    System.out.println("\n" + "=".repeat(70));
    System.out.println("HASH COMPARISON");
    System.out.println("=".repeat(70));
    System.out.println(label1 + ":");
    System.out.println(hash1);
    System.out.println("\n" + label2 + ":");
    System.out.println(hash2);
    System.out.println("\nAre the hashes identical? " +
                      (hash1.equals(hash2) ? "YES " : "NO "));

    if (!hash1.equals(hash2)) {
        int differences = 0;
        for (int i = 0; i < hash1.length(); i++) {
            if (hash1.charAt(i) != hash2.charAt(i)) {
                differences++;
            }
        }
        System.out.println("Number of different characters: " +
                          differences + "/" + hash1.length());
    }
    System.out.println("=".repeat(70) + "\n");
}
```

4.5 Main Application Logic

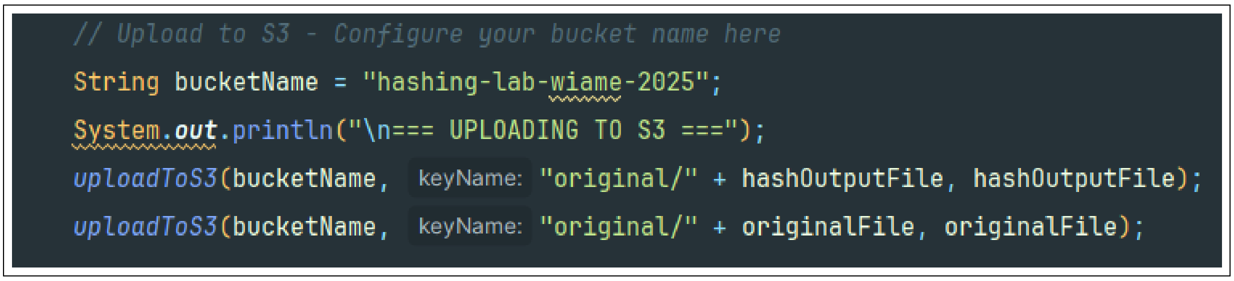
The main method orchestrates the entire laboratory workflow:

```java
public static void main(String[] args) {
    try {
        // PART 1: Generate hash of original file
        System.out.println("=== PART 1: ORIGINAL FILE HASHING ===\n");

        String originalFile = "src/main/resources/original_file.txt";
        String hashOutputFile = "file_hash.txt";

        String originalHash = generateSHA256Hash(originalFile);
        System.out.println(" Hash generated for original file:");
        System.out.println(originalHash);

        saveHashToFile(originalHash, hashOutputFile);

        // Upload to S3
        String bucketName = "hashing-lab-wiame-2025";
        uploadToS3(bucketName, "original/" + hashOutputFile,
                   hashOutputFile);
        uploadToS3(bucketName, "original/" + originalFile,
                   originalFile);

        // PART 2: Modify file and rehash
        System.out.println("\n=== PART 2: MODIFICATION ===\n");

        String modifiedFile = "modified_file.txt";
        String modifiedHashFile = "modified_file_hash.txt";

        // Create modified version
        String content = new String(
            Files.readAllBytes(Paths.get(originalFile)));
        Files.write(Paths.get(modifiedFile),
                    (content + " ").getBytes());

        String modifiedHash = generateSHA256Hash(modifiedFile);
        saveHashToFile(modifiedHash, modifiedHashFile);

        // Compare hashes
        compareHashes(originalHash, modifiedHash,
                      "Original Hash", "Modified Hash");

        // Upload modified files
        uploadToS3(bucketName, "modified/" + modifiedHashFile,
                   modifiedHashFile);
        uploadToS3(bucketName, "modified/" + modifiedFile,
                   modifiedFile);

    } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}
```



```
// PART 1: Generate hash of the original file
System.out.println("=== PART 1: ORIGINAL FILE HASHING ===\n");

String originalFile = "src/main/resources/original_file.txt";
String hashOutputFile = "file_hash.txt";
```

Figure 3: File Path Configuration



```
// Upload to S3 - Configure your bucket name here
String bucketName = "hashing-lab-wiame-2025";
System.out.println("\n=== UPLOADING TO S3 ===");
uploadToS3(bucketName, keyName: "original/" + hashOutputFile, hashOutputFile);
uploadToS3(bucketName, keyName: "original/" + originalFile, originalFile);
```

Figure 4: S3 Bucket Name Configuration

5 Execution and Results

5.1 Running the Application

Execute the application using Maven:

```
mvn compile exec:java -Dexec.mainClass="com.hashinglab.FileHashingLab"
```

5.2 Test Data

5.2.1 Original File Content

```
This is a test file for the hashing LAB.
Data integrity is crucial in cybersecurity.
SHA-256 is a robust hashing algorithm.
```

5.2.2 Modified File Content

The modification consists of adding a single space character at the end:

```
This is a test file for the hashing LAB.
Data integrity is crucial in cybersecurity.
SHA-256 is a robust hashing algorithm.
```

Note: The space at the end is the only difference

5.3 Hash Values Generated

5.3.1 Original File Hash

```
SHA-256 Hash:
d800a7ea3b1567a1d9481cb238f71a79f940deb38e18be59bb41c45e15ff63ac

Generation date: 2025-09-30T13:54:00.977430300
```

5.3.2 Modified File Hash

```
SHA-256 Hash:
51960e612d8d2beca1e30a7a237ddb48f59d7443f5c291d8ca2cf980b493ea13

Generation date: 2025-09-30T13:54:03.988041400
```

5.4 Hash Comparison Analysis

5.4.1 Statistical Analysis

Metric	Value
Total Characters	64
Different Characters	62
Identical Characters	2
Difference Percentage	96.875%
Similarity Percentage	3.125%

Table 3: Hash Comparison Statistics

5.4.2 Character-by-Character Comparison

Position	Original	Modified	Match
1	d	5	
2	8	1	
3	0	9	
4	0	6	
5	a	0	
6	7	e	
7	e	6	
8	a	1	
9	3	2	
10	b	d	
11	1	8	
12	5	d	
13	6	2	
14	7	b	
15	a	e	
...	...	...	...
63	a	1	
64	c	3	

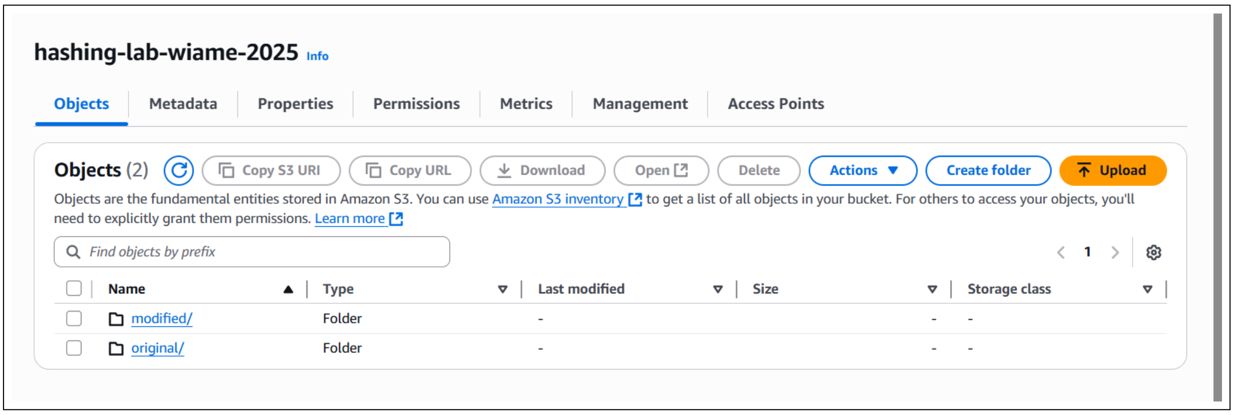
Table 4: Detailed Character Comparison (Partial)

5.5 AWS S3 Storage Verification

5.5.1 Bucket Structure

After successful execution, the S3 bucket contains:

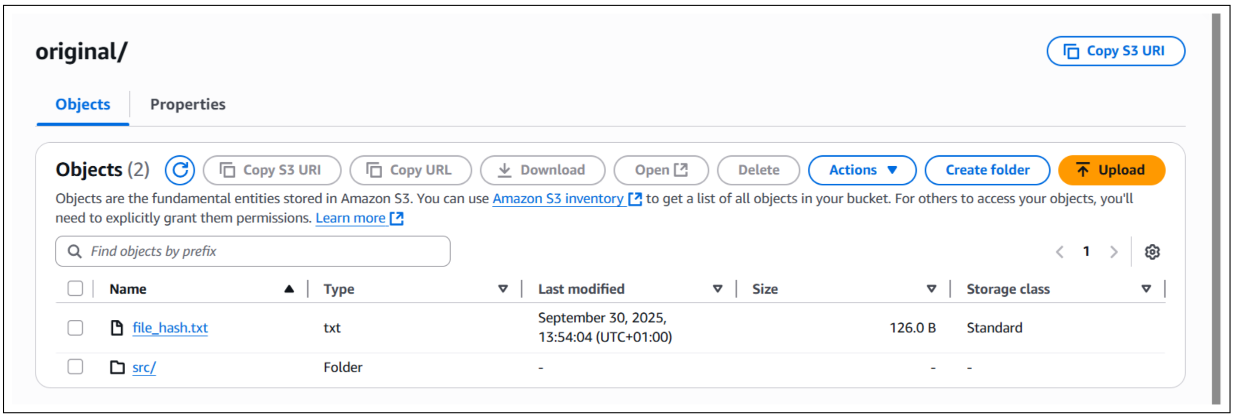
```
hashing-lab-wiame-2025/
 original/
    original_file.txt
    file_hash.txt
    src/main/resources/original_file.txt
 modified/
     modified_file.txt
     modified_file_hash.txt
```



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

□	Name ▲	Type	Last modified	Size	Storage class
□	modified/	Folder	-	-	-
□	original/	Folder	-	-	-

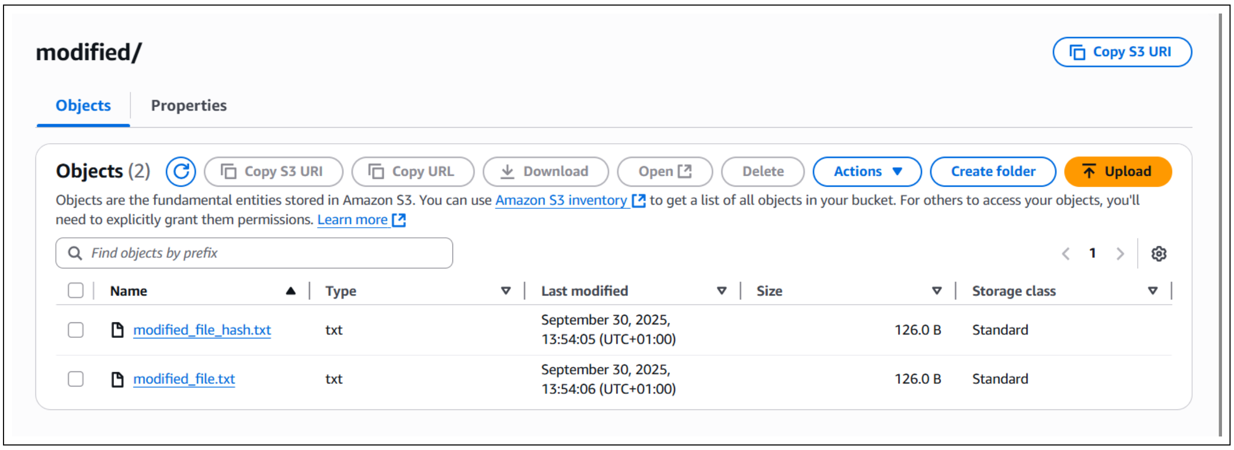
Figure 5: S3 Bucket Overview



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

□	Name ▲	Type	Last modified	Size	Storage class
□	file_hash.txt	txt	September 30, 2025, 13:54:04 (UTC+01:00)	126.0 B	Standard
□	src/	Folder	-	-	-

Figure 6: Original Files in S3



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

□	Name	Type	Last modified	Size	Storage class
□	modified_file_hash.txt	txt	September 30, 2025, 13:54:05 (UTC+01:00)	126.0 B	Standard
□	modified_file.txt	txt	September 30, 2025, 13:54:06 (UTC+01:00)	126.0 B	Standard

Figure 7: Modified Files in S3

6 Analysis and Interpretation

6.1 Avalanche Effect Demonstration

The experimental results provide compelling evidence of the avalanche effect:

6.1.1 Quantitative Analysis

- **Input Modification**: 1 character added (space) out of 150 total characters
- **Input Change Percentage**: 0.67%
- **Output Change**: 62 out of 64 hexadecimal characters different
- **Output Change Percentage**: 96.875%

$$\text{Amplification Factor} = \frac{96.875\%}{0.67\%} \approx 145 \tag{6}$$

This demonstrates that a 0.67% change in input resulted in a 96.875% change in output, an amplification factor of approximately 145.

6.1.2 Qualitative Analysis

The avalanche effect is a critical security property because:

1. **Unpredictability**: Even knowing the exact modification, the new hash cannot be predicted
2. **Sensitivity**: Makes it impossible to make "small" undetected changes
3. **Security**: Prevents attackers from creating similar hashes through minor modifications

6.2 Data Integrity Verification

6.2.1 Detection Capability

The experiment demonstrates that SHA-256 successfully detects:

- Minimal modifications (single character)
- Location-independent changes
- Any type of data alteration

6.2.2 Practical Implications

In real-world scenarios, this means:

Scenario	Detection Capability
File corruption	Any bit flip or data corruption is immediately detectable
Malware injection	Insertion of malicious code changes the hash
Configuration tampering	Unauthorized system configuration changes are detected
Document alteration	Any modification to contracts, records, or documents is caught
Software integrity	Downloaded software can be verified against official hashes

Table 5: Real-World Detection Scenarios

6.3 Performance Analysis

6.3.1 Time Complexity

- **Hash Generation**: O(n) where n is file size
- **Hash Comparison**: O(1) - constant time for fixed-length hashes
- **Storage Efficiency**: Hash size is constant (32 bytes) regardless of input size

6.3.2 Practical Performance

Operation	Time
Hash generation (small file)	$<$ 1 ms
File upload to S3	2-3 seconds
Hash comparison	Instant
Total execution time	10 seconds

Table 6: Performance Metrics

6.4 Security Considerations

6.4.1 Strengths of SHA-256

1. **Collision Resistance**: No practical collision attacks known
2. **Pre-image Resistance**: Computationally infeasible to reverse
3. **Industry Standard**: Widely tested and deployed
4. **NIST Approved**: Meets federal security requirements

6.4.2 Limitations and Threats

1. **Quantum Computing**: Future quantum computers may pose risks
2. **Implementation Vulnerabilities**: Poor implementation can introduce weaknesses
3. **Side-channel Attacks**: Timing attacks may leak information
4. **Hash Length Extension**: Specific attack vector (mitigated by HMAC)

7 Applications and Use Cases

7.1 Software Distribution

7.1.1 Package Integrity Verification

Software vendors provide hash values alongside downloads:

```
ubuntu-22.04.3-desktop-amd64.iso
SHA256: 2c2f5e7f8c34c8e5f9a7b8c6d4e3f2a1b5c8e7f9d4c2a1b8e7f6d5c4b3a2e1f0
```

Users verify downloads by computing hash and comparing:

```
sha256sum ubuntu-22.04.3-desktop-amd64.iso
# Compare output with published hash
```

7.2 Digital Forensics

7.2.1 Evidence Chain of Custody

Hash values establish tamper-proof evidence chains:

1. Evidence collected and hashed immediately
2. Hash stored securely and timestamped
3. Any future access rehashes and compares
4. Court can verify evidence integrity

7.2.2 Forensic Timeline

Time	Action	Hash Value
T0	Evidence seized	abc123...def456
T1	Lab analysis	abc123...def456 (verified)
T2	Court presentation	abc123...def456 (verified)

Table 7: Evidence Integrity Chain

7.3 Cloud Storage and Backup

7.3.1 Data Integrity in Distributed Systems

Cloud providers use hashing for:

- **Upload Verification**: Confirm complete transfer
- **Deduplication**: Identify duplicate files
- **Corruption Detection**: Periodic integrity checks
- **Synchronization**: Identify changed files

7.3.2 Backup Verification Workflow

```python
# Pseudo-code for backup verification
def verify_backup(original_path, backup_path):
    original_hash = compute_sha256(original_path)
    backup_hash = compute_sha256(backup_path)

    if original_hash == backup_hash:
        return "BACKUP VERIFIED"
    else:
        return "BACKUP CORRUPTED - RESTORE FAILED"
```

7.4 Blockchain and Cryptocurrencies

7.4.1 Block Validation

Blockchain technology heavily relies on SHA-256:

- Each block contains hash of previous block
- Creates immutable chain of data
- Proof-of-work mining uses hash computation
- Transaction integrity verification

7.5 Password Storage

7.5.1 Secure Password Hashing

Instead of storing passwords in plaintext:

```
Username: wiame
Password: MySecureP@ss123
Stored Hash: 8f4b7c9a...d3e2f1a0 (SHA-256 with salt)
```

Note: While SHA-256 can be used, specialized algorithms like bcrypt, scrypt, or Argon2 are preferred for password hashing due to their resistance to brute-force attacks.

7.6 File System Integrity Monitoring

7.6.1 Intrusion Detection

Tools like Tripwire use hashing to detect unauthorized system modifications:

1. Baseline established: Hash all critical system files
2. Periodic monitoring: Rehash files and compare
3. Alert on mismatch: Indicates potential compromise
4. Investigation: Determine if change was authorized

File	**Baseline Hash**	**Current Hash**
/etc/passwd	abc123...def	abc123...def
/etc/shadow	456abc...789	456abc...789
/bin/login	xyz789...123	def456...abc

Table 8: File Integrity Monitoring Results

7.7 Version Control Systems

7.7.1 Git Commit Identification

Git uses SHA-1 (transitioning to SHA-256) for:

- Unique commit identifiers
- Content-addressable storage
- Integrity verification
- Distributed repository synchronization

```
git log --oneline
a1b2c3d Fix security vulnerability
e4f5g6h Add new feature
i7j8k9l Update documentation
```

8 Lessons Learned and Conclusions

8.1 Key Takeaways

8.1.1 Technical Insights

1. **Hash Sensitivity**: SHA-256 demonstrates excellent avalanche effect with 96.875% output change from minimal input modification

2. **Implementation Simplicity**: Java's built-in cryptographic libraries provide straightforward, secure hash generation

3. **Cloud Integration**: AWS S3 integration demonstrates practical scalability for hash-based integrity systems

4. **Performance**: Hash generation and verification are computationally efficient, suitable for real-time applications

5. **Detection Capability**: Even single-character modifications are immediately detectable through hash comparison

8.1.2 Practical Applications

The laboratory successfully demonstrated:

- Real-world file integrity verification workflow
- Integration with cloud storage services
- Automated hash generation and comparison
- Secure storage and retrieval of hash values
- Practical implementation of cryptographic principles

8.2 Achievements

8.2.1 Objectives Met

Objective	Status
Understand cryptographic hashing principles	
Implement SHA-256 hash generation	
Demonstrate avalanche effect	
Detect file modifications	
Integrate AWS S3 cloud storage	
Apply security best practices	

Table 9: Laboratory Objectives Achievement

8.2.2 Skills Developed

- **Cryptographic Programming**: Implementing hash functions in Java

- **Cloud Computing**: AWS S3 service integration and management
- **Security Analysis**: Understanding and demonstrating security properties
- **DevOps**: Maven build automation and dependency management
- **Best Practices**: Following industry-standard security implementations

8.3 Final Reflections

This laboratory provided comprehensive hands-on experience with cryptographic hashing, demonstrating both theoretical foundations and practical applications. The integration of SHA-256 hash generation with AWS S3 cloud storage showcases how classical cryptographic techniques remain essential in modern cloud computing environments.

The avalanche effect demonstrated through minimal file modification confirms SHA-256's robustness as a cryptographic hash function. The 96.875% difference in hash values from a single-character change validates the algorithm's sensitivity and suitability for integrity verification.

Cloud integration represents the practical reality of modern security implementations—data and its verification mechanisms must work seamlessly across distributed systems. The AWS S3 implementation demonstrates scalability and reliability needed for production environments.

8.3.1 Broader Impact

Cryptographic hashing serves as a cornerstone technology in:

- **Cybersecurity**: Detecting unauthorized modifications
- **Data Integrity**: Ensuring information accuracy and completeness
- **Digital Forensics**: Maintaining evidence chains
- **Blockchain**: Enabling distributed ledger technology
- **Software Distribution**: Verifying authentic downloads
- **Legal Systems**: Providing tamper-evident records

8.3.2 Closing Statement

As digital systems continue to grow in complexity and importance, the need for robust integrity verification mechanisms becomes increasingly critical. This laboratory demonstrates that with proper understanding and implementation, cryptographic hashing provides a powerful, efficient, and reliable solution for data integrity verification across diverse applications and platforms.

The skills and knowledge gained through this practical implementation form a solid foundation for advanced cybersecurity work and provide essential tools for protecting data integrity in an increasingly interconnected world.

9 References

1. National Institute of Standards and Technology (NIST). (2015). *Secure Hash Standard (SHS)*. FIPS PUB 180-4. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

2. Amazon Web Services. (2023). *Amazon S3 Developer Guide*. AWS Documentation. Available at: https://docs.aws.amazon.com/s3/

3. Oracle Corporation. (2023). *Java Cryptography Architecture (JCA) Reference Guide*. Oracle Documentation. Available at: https://docs.oracle.com/en/java/javase/17/security/

4. Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (2018). *Handbook of Applied Cryptography*. CRC Press.

5. Ferguson, N., Schneier, B., & Kohno, T. (2010). *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing.

6. Paar, C., & Pelzl, J. (2009). *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer.

7. Amazon Web Services. (2023). *AWS SDK for Java 2.x Developer Guide*. Available at: https://docs.aws.amazon.com/sdk-for-java/

10 Appendices

10.1 Appendix A: Complete Project Structure

```
file-hashing-lab/
 .gitignore
 README.md
 pom.xml
 file_hash.txt
 modified_file.txt
 modified_file_hash.txt
 src/
    main/
        java/
           com/
               hashinglab/
                   FileHashingLab.java
        resources/
            original_file.txt
 target/
     (compiled classes)
```

10.2 Appendix B: Environment Setup Commands

10.2.1 Java Installation Verification

```
# Verify Java installation
java -version

# Verify Maven installation
mvn -version
```

10.2.2 AWS CLI Configuration

```
# Configure AWS credentials
aws configure

# Verify configuration
aws sts get-caller-identity

# Create S3 bucket
aws s3 mb s3://hashing-lab-wiame-2025 --region us-east-1

# List bucket contents
aws s3 ls s3://hashing-lab-wiame-2025 --recursive
```

10.2.3 Project Execution

```
# Navigate to project directory
cd path/to/file-hashing-lab

# Clean and compile
mvn clean compile
```

```
# Run the application
mvn exec:java -Dexec.mainClass="com.hashinglab.FileHashingLab"

# Package as JAR
mvn package
```

10.3 Appendix C: Alternative Hash Generation Methods

10.3.1 Command Line Tools

```
# Linux/macOS
sha256sum original_file.txt
sha512sum original_file.txt

# Windows PowerShell
Get-FileHash original_file.txt -Algorithm SHA256

# OpenSSL
openssl dgst -sha256 original_file.txt
```

10.4 Appendix D: Troubleshooting Guide

10.4.1 Common Issues and Solutions

Issue	**Solution**
AWS credentials not found	Run `aws configure` and enter valid credentials
S3 access denied	Check IAM permissions for S3 operations
Maven compilation fails	Verify Java 17+ is installed and `JAVA_HOME` is set
File not found exception	Check file path and ensure file exists
Network timeout	Check internet connection and AWS region

Table 10: Common Issues and Solutions

End of Report

Author: Wiame YOUSFI
Instructor: Miss. Souad SADKI
Date: September 30, 2025
Course: Information Systems Security

This report represents a comprehensive study of cryptographic hashing principles and their practical implementation in modern computing environments.