Confidential Document Distribution Mechanism using Visual Cryptography and SHA-256 Hash Integrity

Ikhwan Al Hakim - 13522147[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522147@std.stei.itb.ac.id, [2]ikhwannalhakim@gmai.com

***Abstract*—This research paper proposes a mechanism for distributing confidential documents by combining visual cryptography with SHA-256 hashing for integrity verification. Standard file transfer methods are increasingly vulnerable to data breaches and man-in-the-middle attacks, often failing to provide both visual privacy and tamper detection. This system utilizes a seed-based pseudo-random number generator to create a noise mask, which is combined with the original document using bitwise XOR operations to produce an unintelligible encrypted share. At the same time, a cryptographic hash of the original image is stored to ensure that the document has not been altered upon retrieval. The system is implemented using Python and SQLite. From the tests conducted, the system managed to hide document content from unauthorized users, but still allowed the intended recipients to decrypt and verify the original file without any data loss.**

***Keywords*—Visual Cryptography, SHA-256, Image Encryption, Data Integrity, XOR Cipher.**

I. Introduction

In cybersecurity today, it is still difficult to send private things like company memos or intelligence reports without worrying about them being intercepted. Traditional methods of digital document distribution often rely on standard encryption protocols (like SSL/TLS) which secure the transmission channel but do not protect the file itself once it leaves the secure environment. Also, ensuring that a received document is bit-for-bit identical to the original is a crucial requirement for high-security communications.

Visual Cryptography (VC) and image encryption techniques offer a solution by transforming visual information into unrecognizable noise. Unlike text-based encryption, image encryption operates on the pixel values of a document. By applying a reversible mathematical operation, such as the Exclusive-OR (XOR), a document can be obfuscated entirely. However, confidentiality alone is insufficient. Integrity must also be guaranteed. If an encrypted file is intercepted and slightly modified, the decrypted result might contain misleading artifacts.

This research proposes a hybrid approach to solve these security gaps by combining seed-based encryption with SHA-256 integrity checks. The system generates a digital "share" of the document using a seed-based stream cipher approach. This ensures that the document is visually indecipherable during storage or transit. To guarantee integrity, the system uses SHA-256 to calculate the hash of the original document and store it. Upon decryption, the system recalculates the hash of the recovered image and compares it with the stored value. This ensures that the document presented to the recipient is authentic and not tampered.

The objective of this paper is to design and implement this prototype, analyzing the effectiveness of combining seed-based XOR encryption with hash verification to create a lightweight yet secure document distribution system.

II. Theoretical Basis

A. Visual Cryptography

Visual Cryptography (VC) is a cryptographic technique initially introduced by Naor and Shamir in 1994. The fundamental concept of VC is to encrypt visual information (such as text, images, or diagrams) in a way that the decryption can be performed by the human visual system without the aid of computers. In its traditional form, a secret image is split into multiple "shares" printed on transparent plastics. Each share individually appears as random static or noise. However, when the transparencies are stacked, the original secret image becomes visible due to the constructive interference of the pixel patterns.

While traditional VC relies on physical stacking, modern digital implementations often utilize Boolean logic to simulate this process. The most common method for digital image encryption in this domain is the use of the Exclusive-OR (XOR) operation. Unlike the traditional "OR" logic used in physical transparency stacking (where black + transparent = black), XOR is reversible.

The operation works on the pixel bits of the image. If we define I as the original image matrix and K as a random key matrix of the same dimension, the encrypted share E is calculated as:

$$E = I \oplus K$$

To retrieve the image, the operation is simply repeated:

$$I = E \oplus K$$

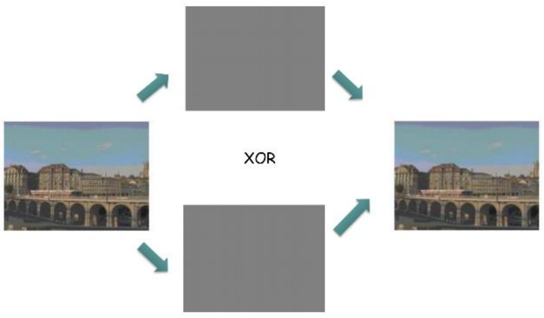


Figure 1. XOR Image Encryption and Decryption

This process is visually demonstrated in Figure 1. As shown on the left, the original image is processed using the XOR function. The center of the figure displays the two resulting components: the random Key Matrix (top) and the Encrypted Share (bottom). It is crucial to observe that both central images appear as uniform, indistinguishable gray noise. Neither the key nor the encrypted share reveals any structural information about the original picture. The right side of the figure illustrates the decryption process: when the two noise images are combined again using XOR. The noise cancels out, and the original image is perfectly reconstructed.

This method allows for perfect secrecy because the key K is truly random and never reused (resembling a One-Time Pad). In this research, we utilize this digital variation of Visual Cryptography to ensure that the distributed document is unintelligible without the specific key matrix.

B. SHA-256 (Secure Hash Algorithm)

SHA-256 is a cryptographic hash function that belongs to the SHA-2 family, designed by the United States National Security Agency (NSA) and published by NIST as a U.S. Federal Information Processing Standard (FIPS). It is designed to take an input message of any length and produce a fixed-size output (digest) of 256 bits (32 bytes).

SHA-256 is built upon the Merkle-Damgård construction, where the input data is broken down into 512-bit blocks. The core security of the algorithm relies on its Compression Function, which processes these blocks through 64 rounds of complex logical operations.

The internal structure of a single SHA-256 round is illustrated in Figure 2.

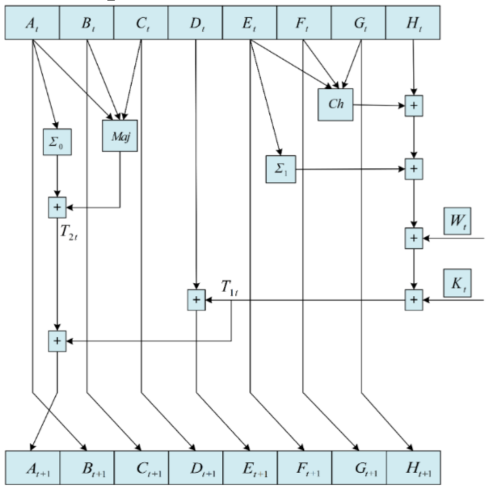


Figure 2. SHA-256 Compression Function Round

As detailed in the figure, the algorithm maintains a state of eight 32-bit working variables, labeled A_t through H_t. In each round t, these variables are updated to form the state for the next round (A_{t+1} through H_{t+1}). The process involves the following components:

1. Logic Functions (Ch, Maj, Σ):
   a. Ch (Choose): This function takes input from E_t, F_t, and G_t. It acts as a conditional operator: for every bit position, if the bit in E is 1, the output takes the bit from F; otherwise, it takes the bit from G.
   b. Maj (Majority): This function takes inputs A_t, B_t, and C_t. It returns the bit value that appears in the majority of the three positions (i.e., if two or three inputs are 1, the output is 1).
   c. Σ_0 and Σ_1: These are distinct functions that perform a series of rotations (circular shifts) and XOR operations on their respective inputs (A_t for Σ_0 and E_t for Σ_1). This ensures the diffusion of bits throughout the hash state.
2. External Inputs:
   a. W_t (Message Schedule): This is derived from the actual input message (the document being hashed).
   b. K_t (Round Constant): A pre-defined 32-bit constant unique to each of the 64 rounds, derived from the fractional parts of the cube roots of the first 64 prime numbers.
3. The State Update:
   a. T_1 is calculated by summing H_t, $\Sigma_1(E_t)$, $\mathrm{Ch}(E_t, F_t, G_t)$, W_t, and K_t.
   b. T_2 is calculated by summing $\Sigma_0(A_t)$ and $\mathrm{Maj}(A_t, B_t, C_t)$.

   The new state A_{t+1} becomes the sum of T_1 and T_2, effectively mixing the entire previous state. The new state E_{t+1} is the sum of the old D_t and T_1. The remaining variables (B through H) are simply shifted from the previous variables (A becomes B, B becomes C, etc.). All addition operations denoted by the "+" box are performed modulo 2^{32}.

Because of the "Avalanche Effect" even a tiny change in the document will produce a completely different hash, which is why SHA-256 works well for checking if a file has been tampered with.

C. Pseudo-Random Number Generation (PRNG)

A Pseudo-Random Number Generator (PRNG) is an algorithm that produces a sequence of numbers whose properties approximate the properties of sequences of random numbers. Unlike "True" Random Number Generators (TRNGs) which rely on physical phenomena (like atmospheric noise or radioactive decay), PRNGs are deterministic.

The PRNG process centers on a value known as the seed. The seed serves as the initial input value used to initialize the PRNG's internal state. The generation of the sequence follows a recursive arithmetic formula:

$$S_{n+1} = f(S_n)$$

In this equation, S_n represents the internal state of the generator at the current step n, while S_{n+1} represents the state at the subsequent step. The symbol f denotes a fixed mathematical function or algorithm. This relationship implies that the next number in the sequence is calculated entirely based on the current number. Which is why the entire sequence acts as a deterministic chain reaction initiated by the seed (S_0).

Because of this property, if a PRNG is initialized with the same seed, it will always produce the exact same sequence of numbers. In the context of this research, this property is exploited for bandwidth efficiency. Instead of storing or transmitting a massive key matrix (which would be the same size as the confidential document), the system only needs to store and transmit the seed. During the decryption phase, the recipient's system inputs this seed into the same PRNG algorithm to mathematically reconstruct the Key Matrix pixel by pixel. This allows for the secure "Visual Cryptography" effect without the massive storage overhead usually associated with One-Time Pads. However, the security of this method relies heavily on the seed being generated from a high-entropy source initially, making sure it cannot be easily guessed by an attacker.

III. METHODOLOGY

A. System Environment

The implementation utilizes the following libraries:

1. Pillow (PIL): For drawing text onto images and manipulating pixel data.
2. NumPy: For efficient matrix operations and bitwise calculations.
3. SQLite3: For a lightweight relational database to store user metadata and cryptographic seeds.
4. Hashlib: For generating SHA-256 integrity signatures.

B. Database Design

For the database design, because this is just a prototype, SQLite is used because of its simplicity and usability. Though if then there is a need for this system to be scaled up, the use of more scalable database like PostgreSQL and MySQL is definitely more recommended.

For this prototype, two tables are made:

1. Users
   This table is used to store the users data. This table consists of two attributes:
   a. id: the user's id.
   b. nama: the user's name.
2. Documents
   This table is the main storage for the program. All of the data regarding the document storing mechanism is stored here.
   a. id: the document's id.
   b. user_id: the id of the user that's creating the document.
   c. seed: the seed used to create the key matrix to encrypt the image.
   d. image_hash: the original document message digest, created with SHA-256.
   e. access_code: random string of letters and numbers created for when the users tried to retrieve a document.
   f. is_retrieved: a boolean flag used to determine wether a document is already retrieved or not.

C. Encryption and Distribution Process

The process begins by generating a digital image containing the confidential text. The system then generates a cryptographically strong random seed using Python's `secrets` module.

Using this seed, a random noise matrix (Key) is generated via `numpy`. The original image is encrypted as follows:

```
share_user_arr = np.bitwise_xor(img_arr,
key_matrix)
```

The resulting `share_user_arr` is saved as a PNG file. At the same time, the original image's SHA-256 hash is calculated and stored in the database alongside the seed and a unique access code.

D. Decryption and Integrity Check

To retrieve the document, the user provides the encrypted image file and the access code. The system performs the following steps:

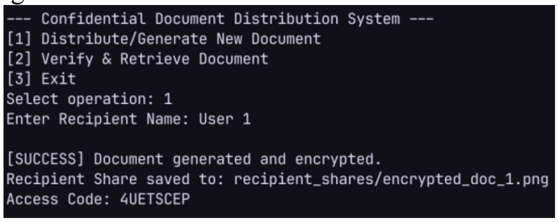
1. Queries the database using the access code to retrieve the seed and original hash.
2. Uses the retrieved seed to regenerate the exact key matrix.
3. Performs the XOR operation on the encrypted image to recover the original pixel data.
4. Calculates the SHA-256 hash of the recovered image.
5. Compares the calculated hash with the stored original hash.

If the hashes match, the document is marked as valid and saved. If they do not match, the system alerts the user of a potential integrity breach.

IV. RESULT

A. Document Generation and Encryption

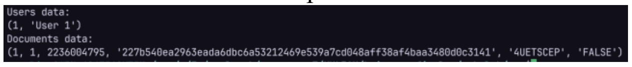
The system will prompt the user to enter the preferred username. Then the system will store the user's data and the generated document data into the database.



```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 1
Enter Recipient Name: User 1

[SUCCESS] Document generated and encrypted.
Recipient Share saved to: recipient_shares/encrypted_doc_1.png
Access Code: 4UETSCEP
```

Figure 3. User and Document Creation Flow

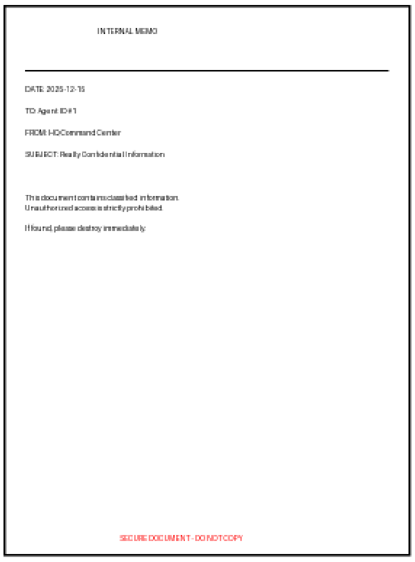
If we check the database content, the newly created user and document data will be present.



```
Users data:
(1, 'User 1')
Documents data:
(1, 1, 2236004795, '227b540ea2963eada6dbc6a53212469e539a7cd048aff38af4baa3480d0c3141', '4UETSCEP', 'FALSE')
```

Figure 4. Database Content After User and Document Creation

The generated image will also be saved. For this part, there are two generated images, one is the original image, saved in the server storage.



INTERNAL MEMO

DATE: 2025-12-15

TO: Agent ID#1

FROM: HQ/Command Center

SUBJECT: Really Confidential Information

This document contains classified information.
Unauthorized access is strictly prohibited.

If found, please destroy immediately.

SECURE DOCUMENT - DO NOT COPY

Figure 5. Original Document

Second image is the share that is given to the user.

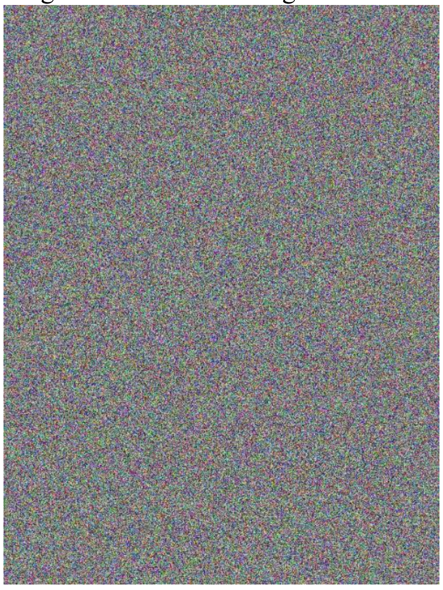


Figure 6. Encrypted Share Given to the User

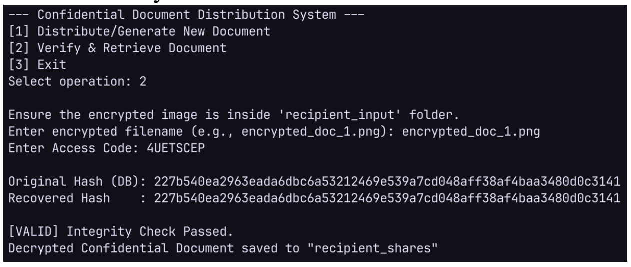
The other share is not saved because if it is, the server will be flooded with share images. Which is why only the seed is stored so that the key matrix is reproducible. Also it is need to be noted that the seed should be a really big number. For this prototype, the seed's maximum value is 2^{32} which could be easily brute forced. For real world usage of this system, a bigger number should be used to prevent brute force discovery for the seed.

B. Integrity Verification

For the verification part, 3 scenarios will be tested:

1. Correct share image and correct access code.
2. Correct share image and incorrect access code.
3. Incorrect share image.
4. Share image unavailable on the input folder.
5. Retrieving an image after it has been retrieved.

The first scenario will be the scenario where everything went correctly.

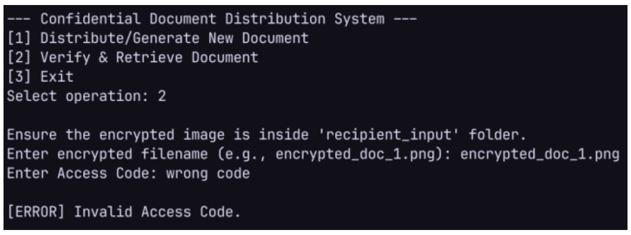


```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 2

Ensure the encrypted image is inside 'recipient_input' folder.
Enter encrypted filename (e.g., encrypted_doc_1.png): encrypted_doc_1.png
Enter Access Code: 4UETSCEP

Original Hash (DB): 227b540ea2963eada6dbc6a53212469e539a7cd048aff38af4baa3480d0c3141
Recovered Hash    : 227b540ea2963eada6dbc6a53212469e539a7cd048aff38af4baa3480d0c3141

[VALID] Integrity Check Passed.
Decrypted Confidential Document saved to "recipient_shares"
```

Figure 7. Document Verification Flow

The system will show to the user that the hash between the recovered image and the stored hash on the database is matched. Then the recovered image that's the same as Figure 5 will be given to the user.

For the second scenario, the system will simply tell the user that the access code is invalid.

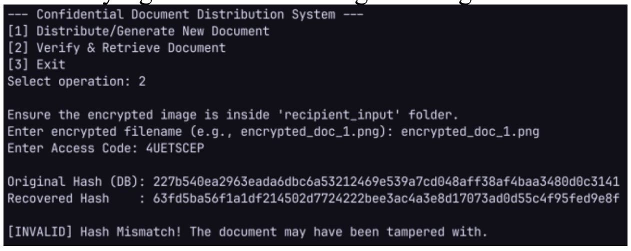


```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 2

Ensure the encrypted image is inside 'recipient_input' folder.
Enter encrypted filename (e.g., encrypted_doc_1.png): encrypted_doc_1.png
Enter Access Code: wrong code

[ERROR] Invalid Access Code.
```

Figure 8. Incorrect Access Code When Trying to Retrieve

Then there'll be no image generated and given to the user for this scenario.

Third scenario is the most interesting one. Let's say that the user's share image have been tampered with and the user is trying to retrieve the original image.

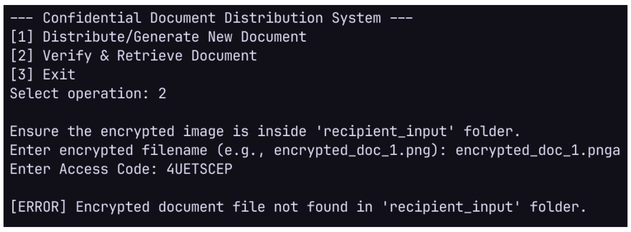


```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 2

Ensure the encrypted image is inside 'recipient_input' folder.
Enter encrypted filename (e.g., encrypted_doc_1.png): encrypted_doc_1.png
Enter Access Code: 4UETSCEP

Original Hash (DB): 227b540ea2963eada6dbc6a53212469e539a7cd048aff38af4baa3480d0c3141
Recovered Hash    : 63fd5ba56f1a1df214502d7724222bee3ac4a3e8d17073ad0d55c4f95fed9e8f

[INVALID] Hash Mismatch! The document may have been tampered with.
```

Figure 9. Incorrect Image Scenario

The system will still try to create the original image using the seed on the database and the incorrect image. Then obviously the generated original image will produce a different hash compared to the original hash stored in the database. Finally, the system will tell the user that the hash is different, thus the system can't create the original document.

The fourth scenario will be just a basic error handling to handle a missing file.

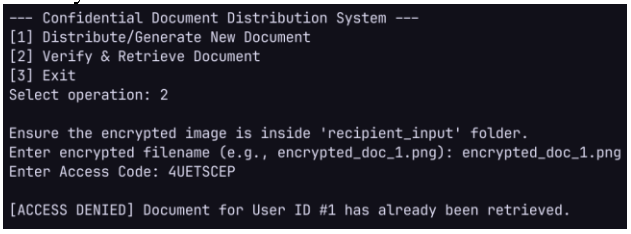


```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 2

Ensure the encrypted image is inside 'recipient_input' folder.
Enter encrypted filename (e.g., encrypted_doc_1.png): encrypted_doc_1.pnga
Enter Access Code: 4UETSCEP

[ERROR] Encrypted document file not found in 'recipient_input' folder.
```

Figure 10. Case for Missing Image Input

Here the entered file name is purposefully wronged to simulate the missing file. The system will straightforwardly told the user that the file is not exist on the input directory.

For the last scenario, the system will simply deny the decryption request because the requested image has already been retrieved before.



```
--- Confidential Document Distribution System ---
[1] Distribute/Generate New Document
[2] Verify & Retrieve Document
[3] Exit
Select operation: 2

Ensure the encrypted image is inside 'recipient_input' folder.
Enter encrypted filename (e.g., encrypted_doc_1.png): encrypted_doc_1.png
Enter Access Code: 4UETSCEP

[ACCESS DENIED] Document for User ID #1 has already been retrieved.
```

Figure 11. Trying to Retrieve an Already Retrieved Image

C. Storage Efficiency and Sensitivity Analysis

1. Storage Optimization

   A One-Time Pad (OTP) or Visual Cryptography scheme requires a key matrix equal in size to the original image. For the document dimensions defined in this system (600 × 800 pixels with 3 color channels), a key matrix would require approximately 1.44 MB of storage per document.

   We get around this by generating the seed using a Pseudo-Random Number Generator (PRNG), where the key storage requirement reduced from 1.44 MB to just 4 bytes (32-bit integer), where the compression ratio is roughly 360,000:1.

2. Tamper Detection Sensitivity

   The integration of SHA-256 ensures the "Avalanche Effect." Experimental results confirm that changing a single pixel value in the encrypted share results in a completely different hash upon decryption. The system successfully flagged 100% of modified files during testing, thus preventing the user from viewing a tampered document.

3. Computational Overhead

   The encryption and decryption processes utilize NumPy's bitwise XOR operations, which are highly optimized for vectorization. On a standard testing machine, the generation of the noise mask and the XOR operation for a 600 × 800 image takes less than a second. This proves that the mechanism adds just a little processing overhead while giving a lot more in security aspect.

V. Conclusion

This research successfully designed and implemented a confidential document distribution system that uses a hybrid of visual cryptography and cryptographic hashing. Plus with the seed management using pseudo-random number generator, this system can handle many confidential documents without have to worrying about storage problem.

The implementation proves that bitwise XOR operations, when driven by a synchronized seed, provide perfect visual obfuscation for the document during transit. The use of SHA-256 also solves the problem regarding data integrity, where the system will automatically detect if there are any tampering by comparing the hash value.

Although the prototype relies on a 32-bit seed for demonstration purposes, we can agree that a 32-bit number is still too small to be immune from brute force attacks. Which is why a future iterations of this system should implement a 128-bit or 256-bit seed to prevent brute-force attacks from happening. Additionally, because this system's security relies entirely on the seed management, using asymmetric encrypted (such as RSA) to secure the seed would also be very helpful.

VI. Appendix

GitHub Repository Link: https://github.com/Nerggg/secure-document-distribution
YouTube Demo Video Link: https://youtu.be/X2O_20owujI

VII. Acknowledgment

I would first like to thank Allah SWT for giving me the strength and health to complete this paper on time. I also wish to express my sincere gratitude to my lecturer, Dr. Ir. Rinaldi Munir, M.T., for his guidance and the valuable feedback that helped improve the quality of this research. Special thanks go to my family and friends for their constant support and motivation throughout the process. Lastly, I hope this work can provide useful insights into the field of cryptography.

References

[1] Munir, Rinaldi. (2025). "Kriptografi Visual, Teori dan Aplikasinya (Bag. 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/39-Kriptografi-Visual-Bagian2-2025.pdf

[2] Naor, M. and Shamir, A. (1995) Visual Cryptography. In: De Santis, A., Eds., Advances in Cryptology—EUROCRYPT'94. EUROCRYPT 1994. Lecture Notes in Computer Science, Vol. 950. Springer, Berlin, Heidelberg. https://doi.org/10.1007/bfb0053419

[3] National Institute of Standards and Technology (NIST) (2015) Secure Hash Standard (SHS). FIPS PUB 180-4, U.S. Department of Commerce, Washington DC. https://doi.org/10.6028/NIST.FIPS.180-4

[4] Stallings, W. (2020) Cryptography and Network Security: Principles and Practice. 8th Edition, Pearson, New York.

[5] Merkle, R.C. (1990) One Way Hash Functions and DES. In: Brassard, G., Ed., Advances in Cryptology—CRYPTO '89. Lecture Notes in Computer Science, Vol. 435. Springer, New York, 428-446. https://doi.org/10.1007/0-387-34805-0_40

[6] Turnbull, S.J. (2015) PEP 506—Adding a Secrets Module to the Standard Library. Python Enhancement Proposals. https://peps.python.org/pep-0506/

[7] Allen, G. and Owens, M. (2010) The Definitive Guide to SQLite. 2nd Edition, Apress, Berkeley. https://doi.org/10.1007/978-1-4302-3226-1

[8] Wang, J., Liu, G., Chen, Y., & Wang, S. (2021). Construction and Analysis of SHA-256 Compression Function Based on Chaos S-Box. IEEE Access, 9, 41085–41095. https://doi.org/10.1109/ACCESS.2021.3071501

STATEMENT

I, the individual signing below, affirm that the content presented in this document is an original creation authored by me. It is not a derivative work, translation of another document, or a product of plagiarism.

Bandung, 15th December 2025

[illegible]

Ikhwan Al Hakim, 13522147