

Implementation of Huffman Encoding for Modified RSA-AES Encrypted Token Compression in Secure Banking Transactions

¹Hamidu M.

mohammedhamidu1987@gmail.com

Department of computer Science, Adamawa State University, Mubi.
Adamawa State, Nigeria

²Sarjiyus O.

Department of computer Science, Adamawa State University, Mubi.
Adamawa State, Nigeria

³Manga I.

Department of computer Science, Adamawa State University, Mubi.
Adamawa State, Nigeria

DOI: 10.56201/ijcsmt.vol.11.no3.2025.pg.98.128

Abstract

This research Huffman encoding for modified RSA-AES encrypted token compression in secure banking transactions aims to improve the security strength of customer banking credentials in transit and at rest by modifying the RSA token generation stage of encryption. These tokens are not original banking credentials but 32-bit decryption keys of AES. This modification will be made possible by using SHA-256 token generation for its historic strength and resistant to brute-force attack. This approach may hinder a serious computational overhead and time-space complexity. However, we propose the use of Huffman encoding with its quicker data compression to overcome the data size intricacy.

Keyword: SHA-256, avalanche effect, Huffmann Encoding, Entropy, overhead

INTRODUCTION

Securing banking transactions have become increasingly challenging in today's digitally connected world where financial data is constantly being transmitted over various networks. The potential risks associated with data breaches and unauthorized access to sensitive information have made encryption and data compression essential components of modern banking systems (Haryaman *et al* 2024). Securing sensitive customer banking tokens like credit card numbers and account credentials is also essential (Agur *et al* 2020). As more transactions and communications occur

digitally, banks and other financial institutions must ensure customer's data is protected during storage and transmission (Javaid *et al* 2022). Currently, many payment networks and banking systems use AES a symmetric encryption standard to protect tokens and data in transit and at rest. AES applies cipher block chaining (CBC) and other techniques to encrypt plain text data into uncomprehensible ciphertexts (Altigani *et al* 2021).

AES is widely used globally to protect classified information (Smid, 2021). AES was chosen to replace the older Data Encryption Standard (DES) which was vulnerable to brute force attacks by National Institute of Standards and Technology (NIST) in 2001 after a 5-year standardization process, it is considered very difficult to crack through brute force attacks. AES transforms plain text data into fixed block sizes of ciphertexts and encryption keys. AES provides very high security against known attacks with its multiple round structure and large secret key sizes. It encrypts and decrypts data in fixed block sizes of 128 bits using cryptographic keys of 128-bits, 192-bits or 256-bits (Kishor Kumar *et al* 2024). It applies substitution, permutation and transformation techniques in multiple rounds to convert plaintext to ciphertext and back. Each round uses different keys derived from the original key using key scheduling algorithms. The number of rounds depends on the key size - 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. The more rounds used, the more secure the AES encryption is against attacks. Analysing AES encrypted data without knowing the original key is extremely difficult given the complexity of reverse engineering the multiple substitution, permutation and transformation rounds. Brute force attacks trying all possible key combinations also become infeasible as key sizes grow larger (Andersson, 2023). No effective cryptographic attacks against AES itself are publicly known so far (Grassi *et al* 2021). The only risk is if inadequately secured keys get compromised. By encrypting all bank transaction data with strong 256-bit or higher AES keys, the data is secured even if intercepted during transmission.

To enhance the encryption process, AES is often used alongside other cryptographic algorithms and compression algorithms, ensuring the secure transformation and exchange of classified information. In the decryption process, the inverse mix columns and inverse shift rows steps are executed first. This is followed by the byte substitution step, which uses the inverse Sub Bytes process to perform the inverse transformation, culminating in inverse multiplication. The final result is the restoration of the original plaintext.

This research aimed to integrate AES and modified RSA (RSA-SHA-256) encryption algorithms for data security and employ lossless Huffman encoding for data compression during transmission in the context of secure banking transactions which will protect sensitive customer information while optimizing speed and storage capacity.

Statement of the Problem

The rapid advancements in digital banking necessitate the deployment of secure and efficient encryption techniques to safeguard sensitive transactional data. One of the prevailing challenges in this domain is the optimization of data compression without compromising security. RSA-AES

encryption is widely recognized for its robustness but it is typically results in increased file sizes, which can hinder transmission efficiency and storage capabilities.

Recent publications highlight the potential of integrating Huffman encoding with RSA-AES encryption to optimize data compression. However, they collectively underscore the need for more comprehensive evaluations and comparisons particularly in terms of computational efficiency and applicability across diverse transaction volumes. These limitations could be addressed by developing and rigorously testing hybrid encryption-compression models that ensure both security and efficiency in real-world environments. A critical examination using Avalanche Effect, Compression Time and Decryption and Decompression Time can the true potential of the proposed system model be realized in secure banking transactions.

Aim and Objectives

The aim of this research is to design, develop, and evaluate a hybrid model for data encryption, using AES-RSA, and data compression, using Huffman encoding, to enhance banking data security, and the objectives are:

- i. To design a modified RSA encryption algorithm
- ii. To develop a hybrid model of data encryption and compression using the modified AES-RSA and Huffman encoding
- iii. To evaluate the performance of the model in comparison to existing models.

LITERATURE REVIEW

Tabassum & Mahmood, (2020), Azharul (2019) propose a dictionary-based compression scheme using 5-bit encoding for each character, effectively reducing storage requirements for natural language text. However, the study lacks detailed evaluation data, quantitative results, and comparisons to other techniques.

Habib *et al* (2020) also discuss a dictionary-based text compression technique using reduced bit encoding. Similar to Tabassum and Azharul's study, this paper lacks detailed evaluation and comparisons, leaving the generalizability to diverse datasets unassessed. Sivanandam, L., Sivanandam *et al* (2020) introduce the Power Transition X Filling and Selective Huffman Coding encoding techniques, which outperform existing methods in compression efficiency. These methods reduce application testing time and memory consumption, though product development constraints affect performance and quality. Kaffah *et al* (2020) investigate the use of AES and Huffman compression for encrypting e-mail messages. The AES-Huffman encryption system achieves high accuracy and performance, but it faces limitations such as vulnerability to hacking and data leakage, with a constraint of 32,200 characters due to compression.

Herzog *et al* (2020) explore the impact of evasive techniques used by Windows malware on antivirus software and possible countermeasures. The study finds that countermeasures can alter malware behavior, but it notes limitations in the analysis of advanced evasion capabilities. Haldar-

Iversen, (2020) examines the use of DEFLATE, dictionary coding, and Huffman coding for ASCII text compression. The study concludes that no method outperforms general-purpose compression programs, with the ACM algorithm achieving better compression ratios for ASCII-heavy texts.

Gajjala *et al* (2020) examine Huffman-based encoding techniques for gradient compression in deep learning, introducing RLH, SH, and SHS encoders. RLH stands out with up to 5.1 times data volume reduction, though computational complexity and efficiency issues hinder widespread quantization technique adoption. Ranjin (2020) presents canonical Huffman coding for image compression using wavelet decomposition and thresholding techniques. The approach efficiently reduces image file sizes by discarding insignificant coefficients and minimizing the codebook size through Huffman coding. Taneja & Shukla, (2021) conduct a comparative study between RSA and an optimized version for enhanced security, emphasizing improved information security and efficiency with reduced resource requirements during encoding and decoding. However, challenges in real-world implementation and scalability remain significant.

Agur *et al* (2020) analyze the growth of digital financial services (DFS) in emerging economies, noting significant increases in digital lending and remittances. However, scaling DFS during crises without proper safeguards exacerbates operational and cyber risks and deepens existing societal divides. Moreover, Sondre (2020) assesses various compression algorithms, including Huffman coding and DEFLATE, in their ability to improve the security and efficiency of data transmission in financial institutions. They conclude that while no single compression method outperforms general-purpose compression programs across all data types,

Wahab *et al* (2021) propose a hybrid approach combining RSA cryptography with Huffman coding and discrete wavelet transform (DWT) for data hiding. The method enhances security and achieves high-quality stego-images, although it lacks comprehensive comparison with other hybrid compression techniques.

Sandhu, (2021) reviews traditional lossless data compression methods, emphasizing the efficiency of Huffman and arithmetic coding validated through simulation. Adaptive methods seek to mitigate the limitations of classical techniques but face challenges in achieving optimal compression efficiency. Bouguessa *et al.* (2021) introduce an adaptive Huffman coding technique combined with chaotic maps for secure data compression. Despite passing NIST randomness tests, the method exhibits slightly lower compression ratios compared to standard techniques, posing increased complexity. Rahman & Hamada, (2023) innovate by combining Burrows-Wheeler transform, GPT-2 language model, and Huffman coding for text compression. However, challenges such as error sensitivity and comparatively lower compression rates affect its broader applicability. Grassi *et al* (2021) explore weak-key distinguishers for AES, extending AES distinguishers to more rounds but acknowledging limitations due to AES key-schedule properties and the complexity of chosen-key distinguishers.

Abhilash *et al* (2023) review the use of RSA and AES encryption methodologies for secure banking, highlighting their effectiveness in preventing security attacks but also noting specific

constraints and challenges in real-world implementation. Paavni G. & Ajay K. (2021) discuss AES image encryption methods, ensuring secure transmission of sensitive data while addressing complexities in managing large image files and real-time encryption requirements.

Recently, several approaches have been proposed for enhancing data security and efficiency in various domains. Prasann *et al.* (2024) conducted an analysis of modern encryption methods, including AES encryption, Huffman Coding, and LSB Steganography. They reported similar findings with a focus on enhancing entropy and the Avalanche Effect, while acknowledging limitations in evaluating specific file types and conducting comprehensive comparisons or computational overhead evaluations. Abdo *et al.* (2024) proposed a hybrid approach to secure and compress data streams within cloud computing environments. Their method aimed to simultaneously enhance data security, reduce storage space requirements, and optimize data transmission speeds. They discussed challenges related to scalability, trade-offs between security and compression efficiency, and computational overhead in resource-constrained cloud environments.

METHODOLOGY

This section examines the conventional RSA (Rivest-Shamir-Adleman) who's amongst its weaknesses are insufficient randomness, V-timing attacks, chosen ciphertext attack (CCA), vulnerability to quantum computing, and large key size requirement. These limitations arose the need for robust, light-weight and reliable hybrid system for data key protection.

The Huffman Algorithm

The basic technique for Huffman encoding involves the following steps:

Step 1: Frequency Calculation

Let $S = \{s_1, s_2, s_n\}$

...
(1Error!
No
sequence
specified.)

be the set of symbols in the input data

Let $f(s_i)$ represent the frequency of symbol s_i

Compute the frequency for each symbol s_i in the dataset.

Step 2: Probability Distribution

The probability of each symbol s_i is given by

$$p(s_i) = \frac{f(s_i)}{\sum_{j=1}^n f(s_j)}$$

...
(2Error!
No
sequence
specified.)

Where $\sum_{j=1}^n f(s_j) = 1$

Step 3: Building the Huffman Tree

Initialize a min-heap H containing all symbols s_i with their frequencies $f(s_i)$.

While there is more than one node in the heap:

Remove the two nodes x and y with the smallest frequencies from the heap.

Create a new node.

$$z \text{ with } f(z) = f(x) + f(y)$$

...
(3Error!
No
sequence
specified.)

Insert z back into the heap.

The final node in the heap is the root of the Huffman tree.

Step4: Assigning Codes

Traverse the Huffman tree to assign binary codes:

Moving left adds a '0' to the code.

Moving right adds a '1' to the code.

The code length $l(s_i)$ for each symbol s_i is determined by its depth in the tree.

Step 5: Encoding

The Huffman code for each symbol s_i minimizes the total cost (expected length) of the encoded data.

The expected length L of the encoded data is given

$$\text{by } L = \sum_{i=1}^n P(s_i) \times l(s_i) \quad \dots (4)$$

Step 6: Optimality

Huffman encoding is optimal for a set of symbols where the goal is to minimize the average code length, satisfying: $H(S) \leq L < H(S) + 1$

Where $H(S)$ is the entropy of the source:

$$H(s) = \sum_{i=1}^n P(s_i) \log_2 p(s_i) \quad \dots (5)$$

The Advanced Encryption Standard (AES)

Step 1: Block and Key Size

AES operates on a **block size** of 128 bits (16 bytes). The **key size** can be 128, 192, or 256 bits.

Let P represent the plaintext block and C the ciphertext block.

Let K represent the encryption key, where K is 128, 192, or 256 bits.

Step2: State Representation

The plaintext P is arranged into a 4x4 state matrix:

$$\text{State} = \begin{pmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{pmatrix}$$

Step3: Key Expansion

The key K undergoes a process called key expansion to generate a series of round keys.

Number of rounds N_r is 10, 12, or 14 for 128, 192, or 256-bit keys respectively.

The round keys W_i is derived using the Rijndael key schedule algorithm.

Step4: Initial Round

AddRoundKey:

Each byte of the state is XORed with the corresponding byte of the initial round key W_0 :

$$\text{State} = \text{State} \oplus W_0 \quad \dots (6)$$

Step5: Main Rounds (Repeated $Nr - 1$ Times)

Each round consists of the following transformations:

SubBytes:

Each byte b in the state matrix is replaced with an entry from an S-Box (substitution box): $b' = S(b)$

ShiftRows:

Rows of the state are shifted cyclically to the left: Row(r) shifted left by r positions.

MixColumns:

Each column of the state matrix is transformed using a fixed polynomial over $GF(2^8)$

$$\begin{bmatrix} c'_0 \\ c'_1 \\ c'_2 \\ c'_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

AddRoundKey:

The state matrix is XORed with the round key for the current round:

$$\text{State} = \text{State} \oplus W_i$$

Step6: Final Round

The final round omits the MixColumns step and only includes:

1. SubBytes
2. ShiftRows

3. AddRoundKey

Step7: Ciphertext Output

After the final round, the state matrix is transformed back into a linear array, resulting in the ciphertext C .

Step8: Decryption

AES decryption involves reversing each step using the inverse operations

Inverse SubBytes, Inverse ShiftRows, Inverse MixColumns, and AddRoundKey with round keys applied in reverse order.

AES relies on complex mathematical structures, such as finite field arithmetic in $GF(2^8)$ and the use of S-Boxes for non-linearity, making it resistant to various forms of cryptanalysis.

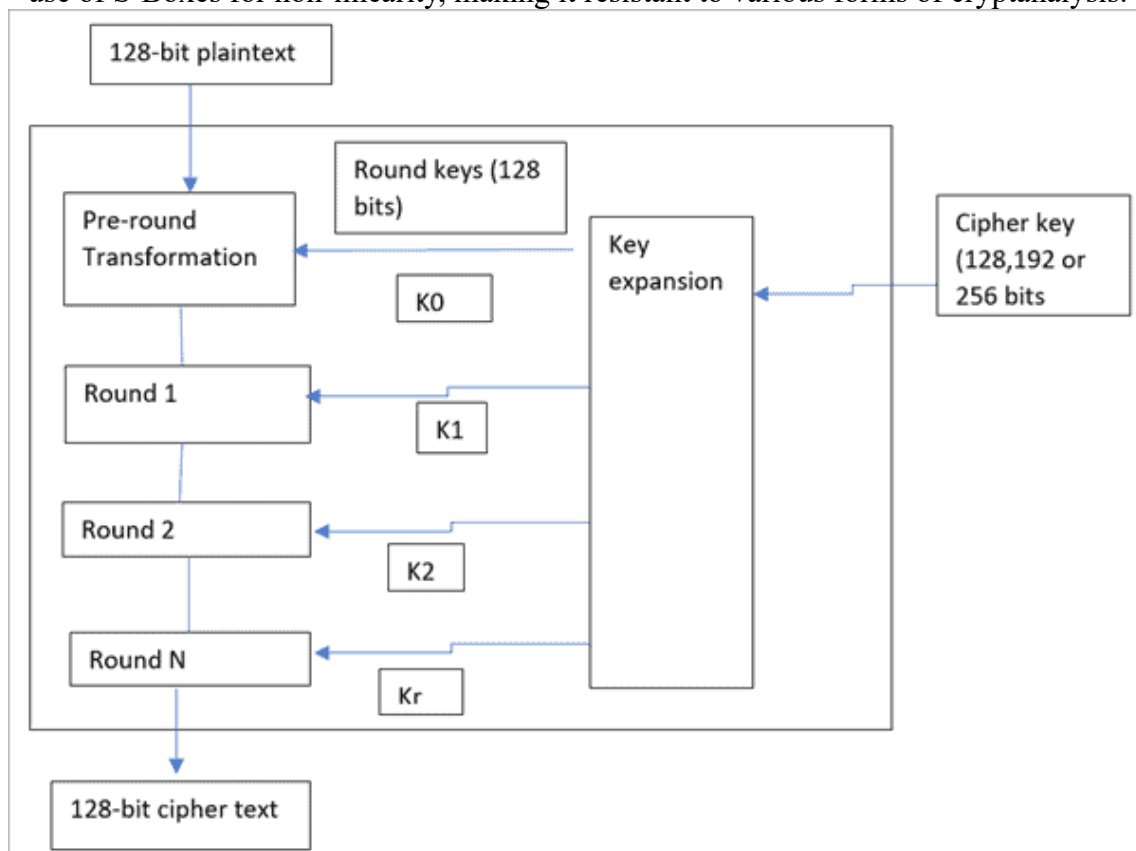


Figure 1: AES Encryption process Structure (Sruthy, 2024)

The Conventional Rivest Shamir Adleman (RSA)

Here's a detailed explanation of the encryption and decryption process.

Step 1: Key Generation

Prime Selection:

Choose two large prime numbers p and q .

Modulus Calculation:

Compute n , the modulus for both the public and private keys

$$n = p \times q \quad \dots (1)$$

Totient Calculation:

Calculate the totient $\phi(n)$ (Euler's totient function) for n

$$\phi(n) = (p - 1) \times (q - 1) \quad \dots (2)$$

Public Key e :

Choose an integer e such that

$1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$ (i.e., e is coprime to $\phi(n)$).

Private Key d :

Calculate d , the modular multiplicative inverse of e modulo $\phi(n)$

$$D \equiv e^{-1} \pmod{\phi(n)} \quad \dots (3)$$

This means d satisfies: $(d \times e) \pmod{\phi(n)} = 1$

Keys:

Public Key: (e, n)

Private Key: (d, n)

Step 2: Encryption Process

Convert Message to Integer (m)

Transform the plaintext message into an integer m such that $0 \leq m < n$. This can be done using various encoding schemes like UTF-8.

Use the public key (e, n) to compute the ciphertext

$$C = M^e \pmod{n} \quad \dots (5)$$

Step 3: Decryption Process

Use the private key (d, n) to retrieve the original message m

$$M = C^d \pmod{n} \quad \dots (6)$$

Convert Integer Back to Message

Decode the integer m back to the original plaintext message using the same encoding scheme applied during encryption.

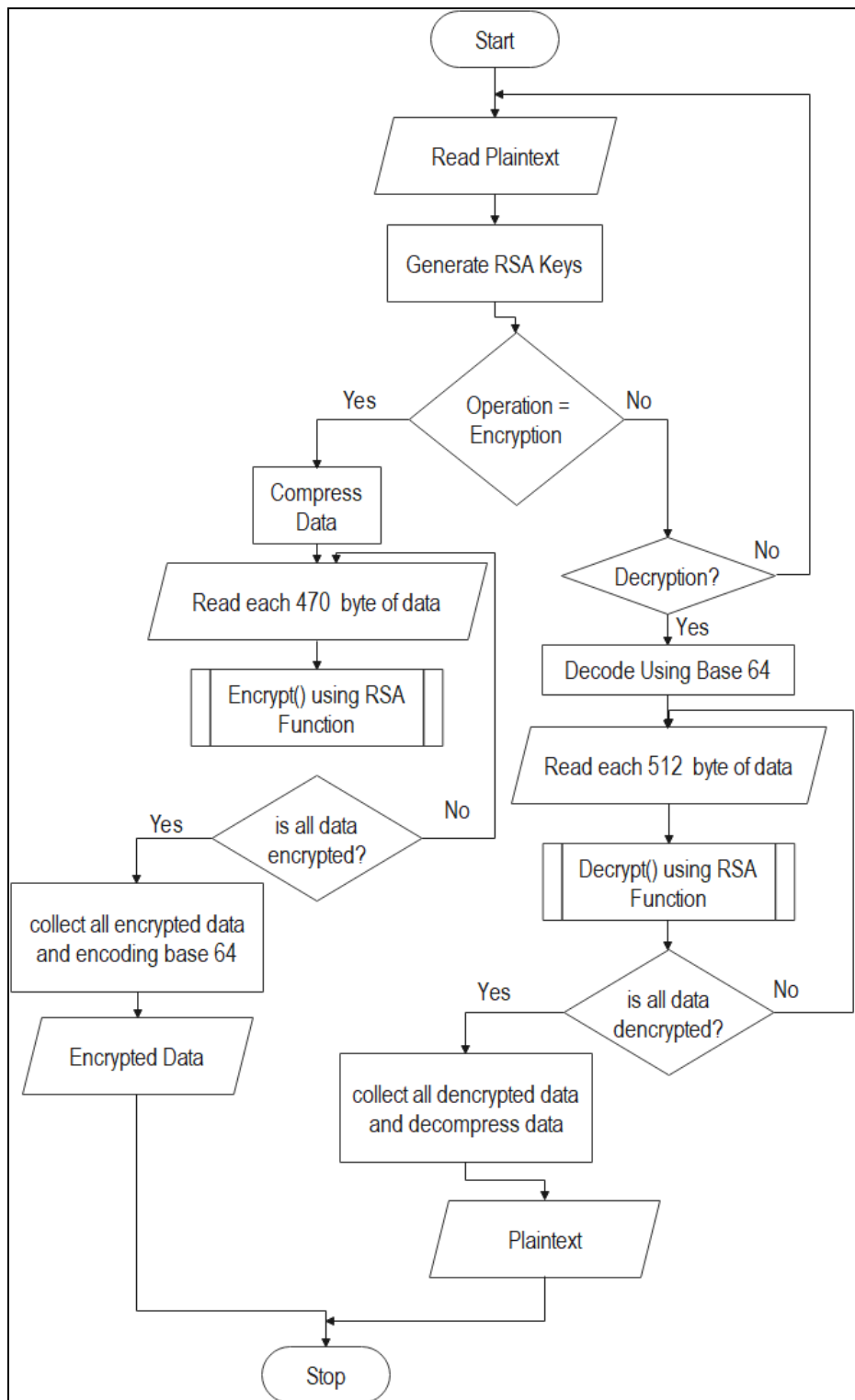


Figure 2: RSA Flowchart (Imam et al., 2022).

RSA is a widely used asymmetric encryption algorithm but despite its robustness in secret key management, RSA has several security weaknesses which include: large key size requirement, vulnerability to quantum computing, chosen ciphertext attack (CCA), timing attack, and insufficient randomness. These shortcomings raise alarm of serious concern on strengthening the algorithm to keep off such threatening threats precisely its vulnerability to quantum computing.

RSA is susceptible to quantum computing attacks, specifically Shor's algorithm, which can factor large integers efficiently, potentially breaking RSA encryption.

The proposed system

The proposed hybrid system for data security that integrate compression and encryption techniques to enhance data security will use Advanced Encryption Standard (AES) to encrypt the data. To securely distribute the AES key, it is encrypted using the Rivest-Shamir-Adleman (RSA) algorithm. This layered approach addresses RSA's vulnerability to large data sizes by limiting its use to encrypting only the AES key, not the entire data. The system then uses Huffman encoding to compress the cyphertexts for transmission. By compressing the data before encryption, this system reduces the amount of data being processed, enhancing efficiency and security. AES provides fast and secure data encryption, while RSA securely manages key exchange its shortcomings mentioned in section 3.3.3 is anticipated to be overcome by hashing the message instead of signing the entire message directly using a secure hash function (SHA-256). The resulting hash value is then signed using the RSA private key. This will absolutely eliminate the chance of brute-force attack

System Design

Firstly, the conventional RSA is modified in the following steps;

Modified Rivest Shamir Adleman (RSA)

The process of hashing the message before encrypting

Step 1: Hash the Message

Compute the hash h of the message M

$$h = \text{Hash}(M) \quad \dots (1)$$

Step 2: Sign the Hash

Encrypt the hash using the RSA private key d

$$S = h^d \bmod n \quad \dots (2)$$

Step4: Verification

To verify, the recipient decrypts the signature using the RSA public key e to recover the hash, then compares it with the hash of the received message

$$h' = S^e \bmod n \quad \dots (3)$$

If h' matches Hash (M), the signature is valid.

Proposed Hybrid Algorithm

Here is an algorithm integrates AES, RSA with hashing, and Huffman encoding into a single hybrid cryptographic algorithm.

Step 1: Data Encryption with AES

Generate AES Key: Generate a random symmetric key K_{AES} for AES encryption.

Encrypt Data: Encrypt the plaintext data P using AES with the generated key K_{AES}

$$CAES = AES_Encrypt(K_{AES}, P) \quad \dots (4)$$

Step 2: Encrypt AES Key with Hashed RSA

Hash the AES Key: Hash the AES key K_{AES} using a secure hash function (SHA-256) to get a fixed-length representation

$$H(K_{AES}) = Hash(K_{AES}) \quad \dots (5)$$

RSA Encryption: Encrypt the hashed AES key $H(K_{AES})$ using the RSA public key (e, n)

$$C_{RSA} = (H(K_{AES}))^e \bmod n \quad \dots (6)$$

C_{RSA} is the RSA-encrypted hash of the AES key.

Step 3: Compress Tokens Using Huffman Encoding

Concatenate Ciphertext and RSA Token: Combine C_{AES} and C_{RSA} into a single message for transmission.

$$T = \text{CAES} \parallel \text{CRSA} \quad \dots (7)$$

\parallel denotes concatenation

Huffman Encoding: Compress the concatenated tokens T using Huffman encoding

$$T_{\text{compressed}} = \text{Huffman_Encode}(T) \quad \dots (8)$$

Step 4: Decryption Process on the Recipient's Side

Huffman Decoding:

Decompress the received data using Huffman decoding to retrieve the concatenated tokens T : $T = \text{Huffman_Decode}(T_{\text{compressed}})$

Extract Components:

Split T into C_{AES} and C_{RSA}

RSA Decryption:

Decrypt C_{RSA} the RSA private key (d, n) to retrieve the hashed AES key:

$$H(\text{KAES}) = (C_{\text{RSA}})^d \bmod n \quad \dots (9)$$

AES Key Recovery:

Since the original algorithm encrypts a hash, you must either:

Derive the AES key from the decrypted hash if a deterministic process was used during key generation.

If a non-deterministic hash was used, store a mapping securely to retrieve the original key

AES Decryption:

Decrypt C_{AES} using K_{AES} to recover the original plaintext data

$$P = \text{AES_Decrypt}(K_{\text{AES}}, C_{\text{AES}}) \quad \dots (10)$$

Finally, the algorithm entails data encryption (Use AES to encrypt the data with a randomly generated key), key encryption (Encrypt the hash of the AES key using RSA) and compression (Compress the resulting ciphertext and RSA-encrypted hash using Huffman encoding)

System Architecture

Figure 3.3 shows the proposed system architecture.

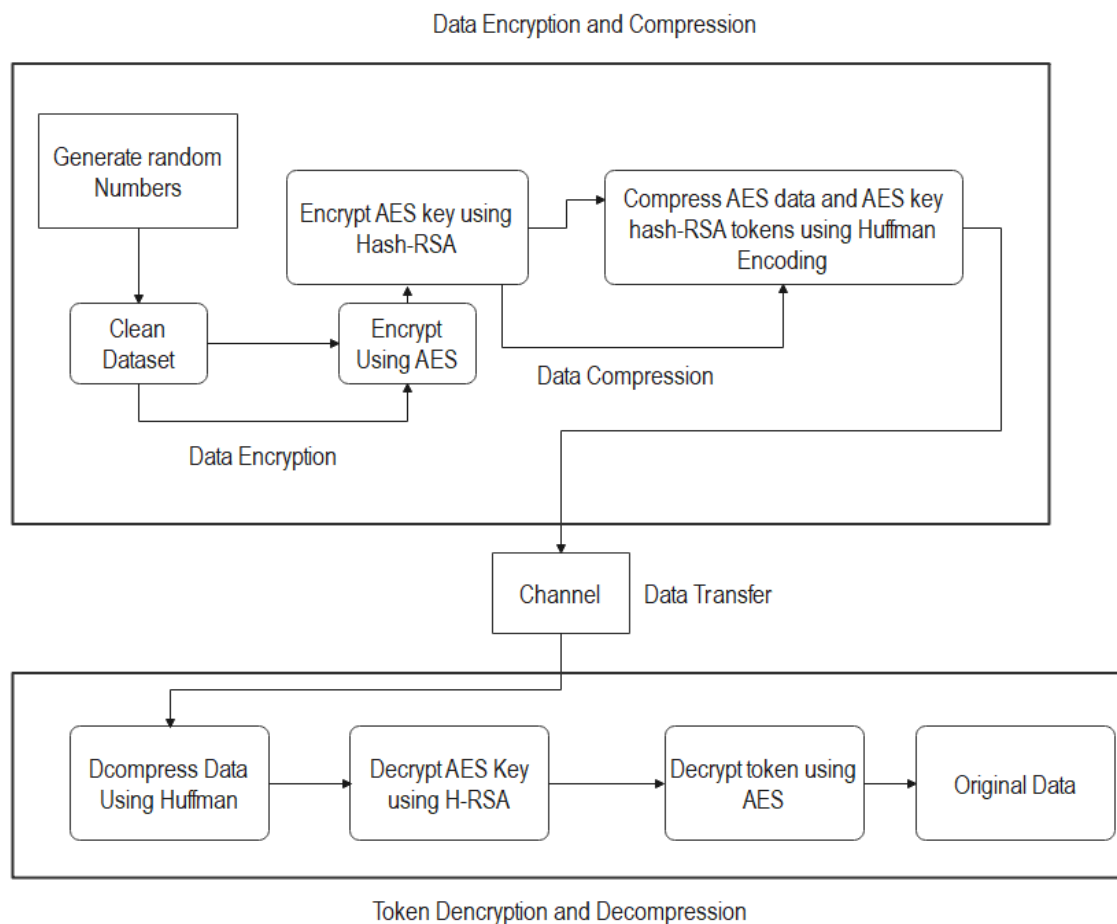


Figure 3: Proposed System Architecture

Working Description of the Model

The first stage of the model operation is data encryption which starts from generating AES keys associated with its random sample data. The next stage is Encrypting the Data using the AES key,

we encrypt the actual data you want to protect. This process turns the data into scrambled information that can only be read if you have the key.

Before we encrypt the AES key, we run it through a hash function (sha-256). Hashing is like creating a unique fingerprint for the key. It ensures the key is represented in a fixed and secure way. Now we take this hashed version of the AES key and encrypt it using RSA a different kind of encryption method that uses a pair of keys, one for encryption (public key) and one for decryption (private key). Only the recipient who has the private key can decrypt and access the AES key.

Compressing the Data Using Huffman Encoding is the next stage which we start by combining everything. After encrypting the data with AES and the key with RSA we put them together into one package and to make the package smaller and easier to send, we use Huffman encoding, a compression technique that reduces the size of the data without losing any information.

The final compressed and encrypted package is sent to the recipient. It's secure, compact, and ready for transmission.

While the data has been received, the recipient first decompresses the package to get back the combined encrypted data and the encrypted AES key. Using their RSA private key, the recipient decrypts the AES key's hashed version. They then use this decrypted information to retrieve the original AES key. Finally, they use the AES key to decrypt the original data, turning it back into a readable format.

Model Evaluation Summary

Evaluating the proposed system involves several structured steps that focus on performance, security, and efficiency. Here's an outline of these steps:

Step 1: Implementation of Encryption and Compression Algorithms:

As previously designed in section 3.5, the process begins by implementing the AES (Advanced Encryption Standard) algorithm for data encryption. AES is typically used for its strong security properties and efficiency.

Integrate a modified RSA algorithm to secure the AES key, ensuring the encrypted tokens are well-protected against unauthorized access.

Apply Huffman encoding to compress the AES-RSA encrypted tokens, aiming to reduce data size for transmission and storage efficiency.

Step 2: Data Collection:

Use a set of randomly generated strings as test data. For consistency, Python's Mersenne Twister library (a widely used random number generator) can be employed to generate these strings, which will simulate sensitive transaction data that the system is designed to secure and compress.

Step 3: Performance Metrics Evaluation:

Compression Ratio:

Measure the efficiency of the Huffman encoding by calculating the compression ratio, which indicates how much the data size has been reduced.

Encryption and Compression Time

Track the time taken by the system to perform both encryption (AES and RSA) and compression (Huffman encoding) steps. This metric is crucial for evaluating the real-time feasibility of the system.

Decryption and Decompression Time

Evaluate the time needed for decrypting and decompressing the data, ensuring that the retrieval of original data remains efficient.

Step 4: Security Analysis:

Conduct a Ciphertext Analysis to check if the data remains secure and unreadable without the necessary keys. AES's strength, alongside RSA's public-key encryption, should be validated for resilience against brute force and chosen ciphertext attacks.

Avalanche Effect Measurement: Assess the avalanche effect, where a small change in the input should result in significant changes in the output. A strong avalanche effect indicates robust encryption.

Entropy Calculation: Calculate the entropy of the encrypted data to ensure high randomness, which contributes to security. High entropy values indicate that the encryption process makes it difficult for attackers to deduce the original data.

Computational Overhead Analysis:

Analyze the computational resources used by the system, particularly memory and processor utilization. This analysis is essential for understanding the scalability of the system, especially for large datasets or high-frequency transaction environments.

Step 5: Comparative Performance Analysis:

Benchmark the proposed system against existing encryption-compression models (if available). This involves comparing metrics such as compression ratio, processing time, and resource utilization to determine the effectiveness of the hybrid AES-RSA-Huffman model over alternatives.

These steps will provide a comprehensive evaluation of the proposed hybrid encryption-compression system, ensuring its effectiveness, security, and practicality in banking applications.

RESULT

This chapter presents and discusses the result and findings of the research. Various input text sizes ranging from 1 kilobyte 1 megabyte were used as input data to test the efficiency of the system.

Figure 4 Authentication Page where user will enter his preferred password for validation

Figure 5 is the AES encrypted password which is the first stage of password token encryption using advanced encryption standard

Figure 6 shows the 32-bit AES key encrypted using modified RSA with SHA-256.

Figure 7 shows the Huffmann Binary encoded data to be stored to be converted then stored for transmission.

Figure 8 shows file encryption-compression time and space report for the modified AES RSA and Huffman hybrid system.

Table 9 shows the encryption space complexity for the conventional AES RSA and Huffman and modified system.

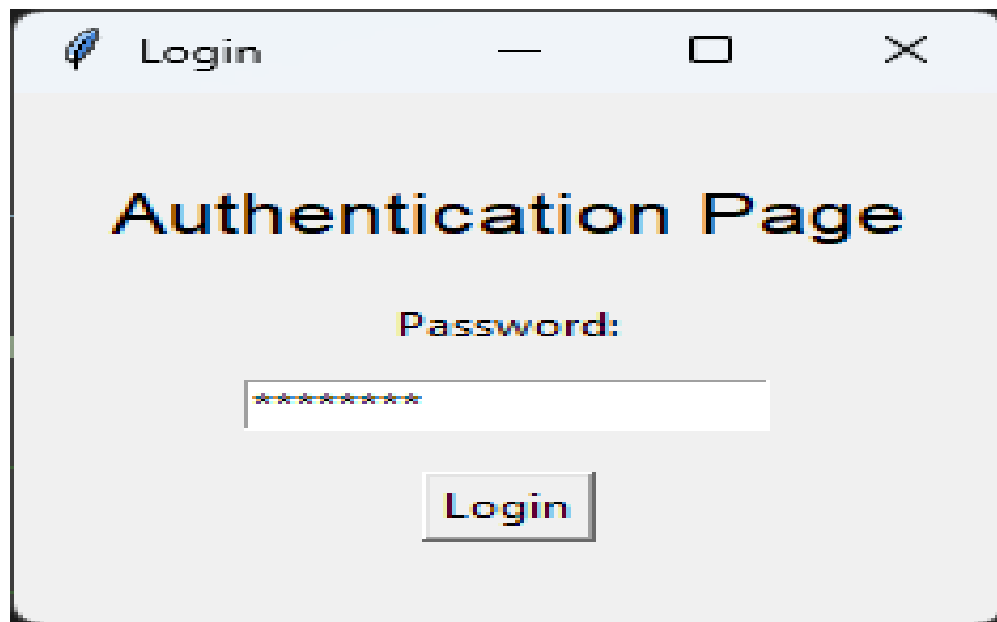
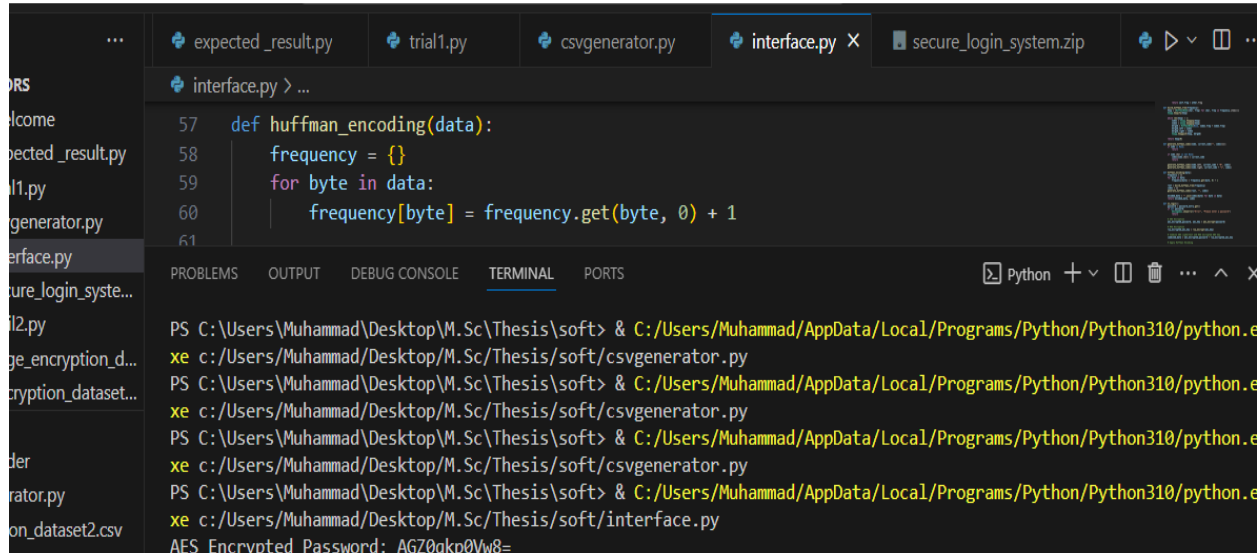


Figure 4: Authentication Page

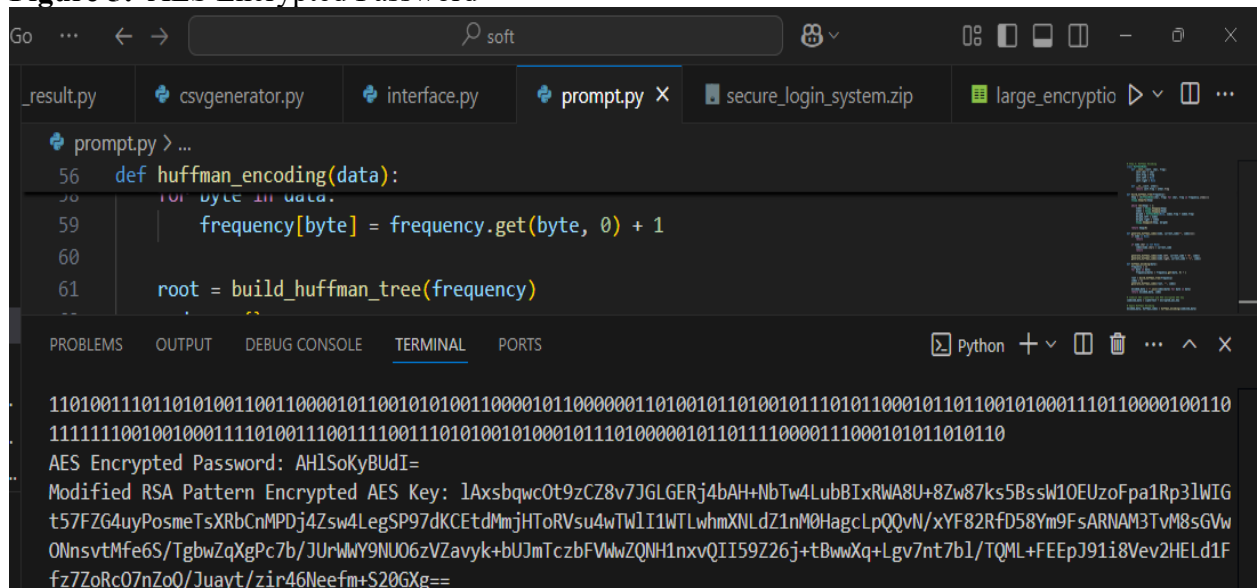


The screenshot shows a Python IDE with several tabs open: 'expected_result.py', 'trial1.py', 'csvgenerator.py', 'interface.py', and 'secure_login_system.zip'. The 'interface.py' tab is active, displaying a Huffman encoding function. The code defines a 'huffman_encoding(data)' function that calculates the frequency of each byte in the data and builds a Huffman tree. The terminal window shows the execution of the program, displaying the AES Encrypted Password: AGZ0qkp0Vw8=.

```
def huffman_encoding(data):  
    frequency = {}  
    for byte in data:  
        frequency[byte] = frequency.get(byte, 0) + 1  
    root = build_huffman_tree(frequency)
```

Terminal Output:
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe
xe c:/Users/Muhammad/Desktop/M.Sc/Thesis/soft/csvgenerator.py
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe
xe c:/Users/Muhammad/Desktop/M.Sc/Thesis/soft/csvgenerator.py
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe
xe c:/Users/Muhammad/Desktop/M.Sc/Thesis/soft/csvgenerator.py
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe
xe c:/Users/Muhammad/Desktop/M.Sc/Thesis/soft/interface.py
AES Encrypted Password: AGZ0qkp0Vw8=

Figure 5: AES Encrypted Password

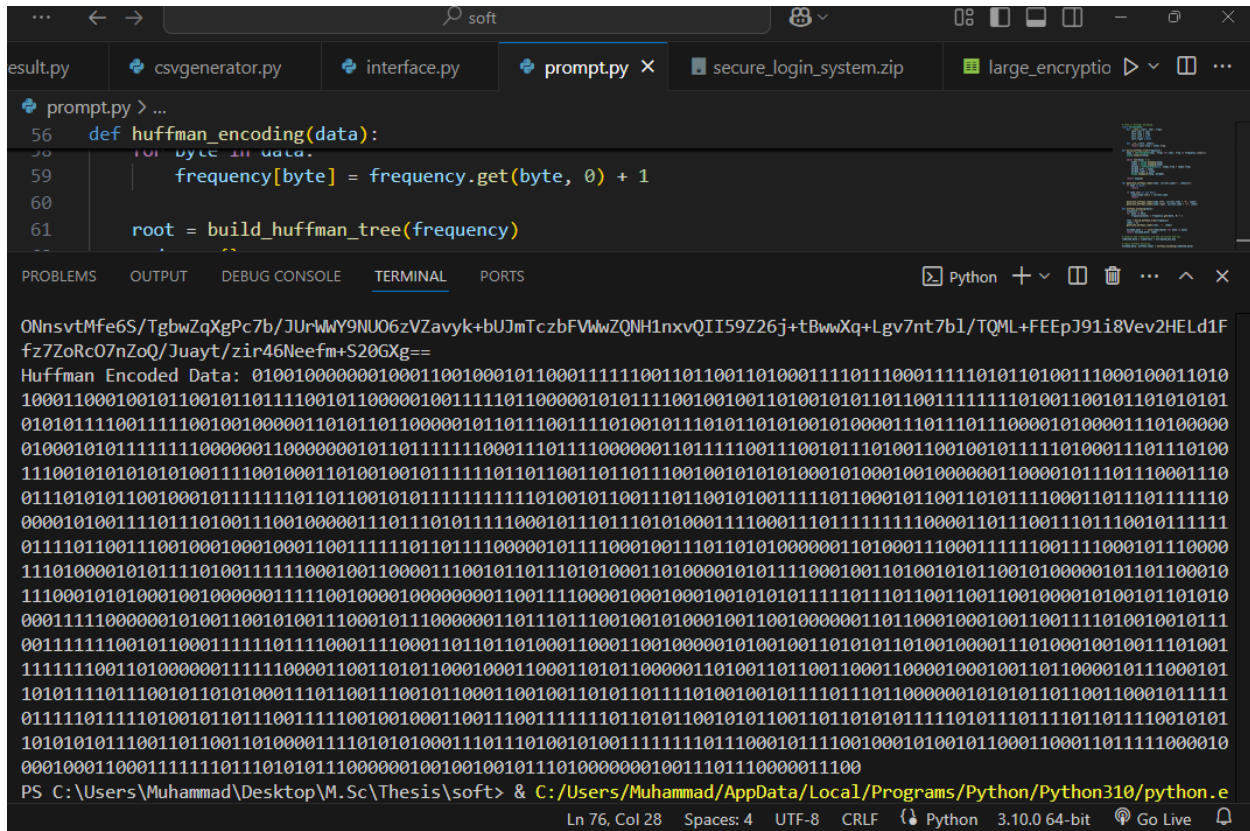


The screenshot shows a Python IDE with several tabs open: 'expected_result.py', 'csvgenerator.py', 'interface.py', 'prompt.py', 'secure_login_system.zip', and 'large_encryptio'. The 'prompt.py' tab is active, displaying a Huffman encoding function. The code defines a 'huffman_encoding(data)' function that calculates the frequency of each byte in the data and builds a Huffman tree. The terminal window shows the execution of the program, displaying the AES Encrypted Password: AHlSoKyBUdI= and the Modified RSA Pattern Encrypted AES Key: 1AxsbcwC0t9zCZ8v7JGLGERj4bAH+NbTw4LubBIxRWA8U+8Zw87ks5Bssw10EUzoFpa1Rp3lWIGt57FZG4uyPosmeTsXRbCnMPDj4Zsw4LegSP97dKCetdMmjHTorVsU4wTWl1I1WlWhmXNLdZ1nM0HagcLpQQvN/xYF82RfD58Ym9FsARNAM3TvM8sGVwONnsvtMfe6S/TgbwZqXgPc7b/JUrwWY9NU06zVZavyk+bUJmTczbFVWwZQNH1nxvQII59Z26j+tBwwXq+Lgv7nt7b1/TQML+FEepJ9118Vev2HEld1Ffz7ZoRc07nZoQ/Juayt/zir46Neefm+S20GXg=.

```
def huffman_encoding(data):  
    frequency = {}  
    for byte in data:  
        frequency[byte] = frequency.get(byte, 0) + 1  
    root = build_huffman_tree(frequency)
```

Terminal Output:
110100111010100110011000010110010101001100000110100101101001011010110001011011001010001110110000100110
1111111001001000111010011100111100111010100101000101101100000101101110000111000101011010110
AES Encrypted Password: AHlSoKyBUdI=
Modified RSA Pattern Encrypted AES Key: 1AxsbcwC0t9zCZ8v7JGLGERj4bAH+NbTw4LubBIxRWA8U+8Zw87ks5Bssw10EUzoFpa1Rp3lWIG
t57FZG4uyPosmeTsXRbCnMPDj4Zsw4LegSP97dKCetdMmjHTorVsU4wTWl1I1WlWhmXNLdZ1nM0HagcLpQQvN/xYF82RfD58Ym9FsARNAM3TvM8sGVw
ONnsvtMfe6S/TgbwZqXgPc7b/JUrwWY9NU06zVZavyk+bUJmTczbFVWwZQNH1nxvQII59Z26j+tBwwXq+Lgv7nt7b1/TQML+FEepJ9118Vev2HEld1F
fz7ZoRc07nZoQ/Juayt/zir46Neefm+S20GXg=

Figure 6: Modified RSA Pattern Encrypted AES Key



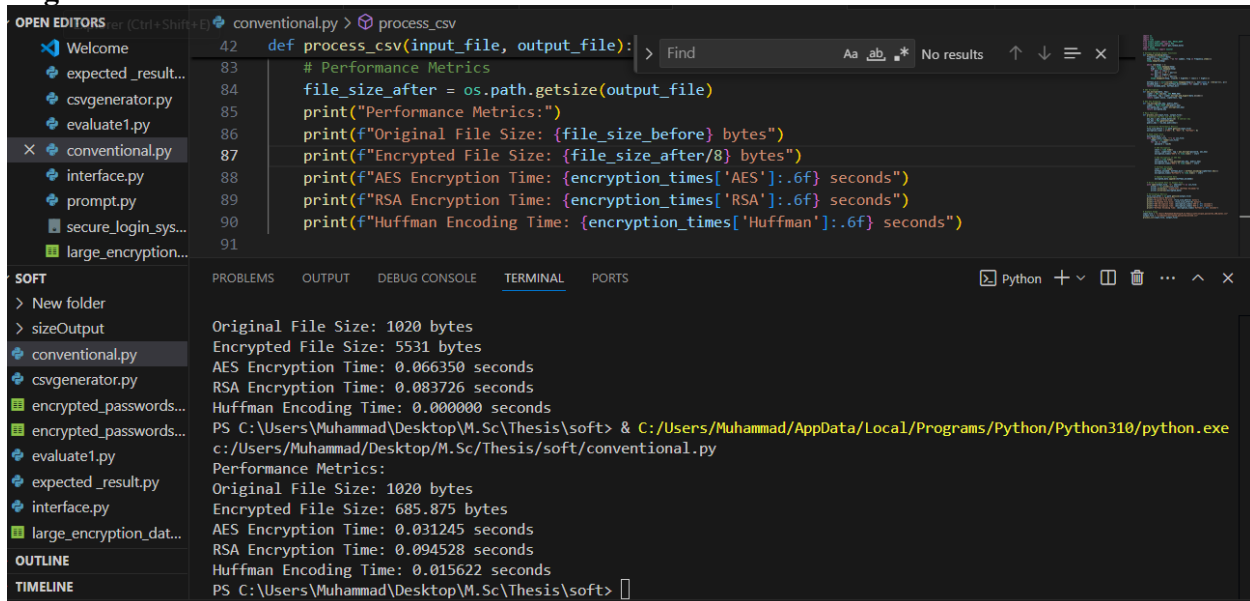
The screenshot shows a Python IDE with a file named `prompt.py` open. The code defines a `huffman_encoding` function. The terminal output displays the Huffman encoded data as a long string of 0s and 1s. The command prompt shows the execution of the script.

```
def huffman_encoding(data):  
    for byte in data:  
        frequency[byte] = frequency.get(byte, 0) + 1  
  
    root = build_huffman_tree(frequency)
```

```
ONsvtMf6S/TgbwZqXgPc7b/JUrWwY9NU06zVZavyk+bUJmTczbFVWwZQNH1nxvQII59Z26j+tBwwXq+Lgv7nt7bl/TQML+FEpJ91i8Vev2HELd1F  
fz7ZoRc07nZoQ/Juayt/zir46Neefm+S20GXg==  
Huffman Encoded Data: 0100100000001000110010001011000111111001101100110100011110111000111101101100111000100011010  
10001100010010100101101111001011000001001111011000001011110010010011010110011111110100110010101010101  
010101110011110010010000011010110110000010110111010010111010110100101000011011101100001010000110100000  
01000101011111100000011000000101101111100011011100000101111001100101101001100100101111010001101110100  
11100101010101001111001000110100100101111101101100110110110010010101000101000100100000110000101110110001110  
0111010101100100010111110110110010111111101001011001101100100111101100010110011011110001101101111110  
000111110000010100110010100111000101110000011011101101100101000100110010000001101100010001001101110110011111  
01110110011100100010001000110011111011011100000101110001001110110100000011010001110001111100111100010110000  
11101000010101110100111110001001100011100101110101000110100010101110001001101001010110010100000101101100010  
11100010101000100100000111110010001000000011001111000100010010101111011101100110011001000010100101101010  
00011111000001010011001010011100010111000001101110111001010001001100100000011011000100010011001110100100111  
0011111100101100011111011100011100011011011010001100011001000001010010011010110100100001110100010010011101001  
111111001101000001111100001100110110001000110011011000001101001100110011001101100010111000101  
1010111011011001011010100011101100111000110010011010111010010010111011101100000010101101100110001011111  
011110111101001011011100110010010001100111111011010101100110110101011110101111010111101011101101110010101  
1010101011100110110011010000111010101000111011010010100111111101100010111001000101100110001010010101  
00010001100011111101101011100000010010010111010000001001110110000011100
```

```
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe
```

Figure 7: Huffman Encoded Data:



The screenshot shows a Python IDE with a file named `conventional.py` open. The code defines a `process_csv` function. The terminal output displays performance metrics for encryption and compression. The command prompt shows the execution of the script.

```
def process_csv(input_file, output_file):  
    # Performance Metrics  
    file_size_after = os.path.getsize(output_file)  
    print("Performance Metrics:")  
    print(f"Original File Size: {file_size_before} bytes")  
    print(f"Encrypted File Size: {file_size_after/8} bytes")  
    print(f"AES Encryption Time: {encryption_times['AES']:.6f} seconds")  
    print(f"RSA Encryption Time: {encryption_times['RSA']:.6f} seconds")  
    print(f"Huffman Encoding Time: {encryption_times['Huffman']:.6f} seconds")
```

```
Original File Size: 1020 bytes  
Encrypted File Size: 5531 bytes  
AES Encryption Time: 0.066350 seconds  
RSA Encryption Time: 0.083726 seconds  
Huffman Encoding Time: 0.000000 seconds  
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft> & C:/Users/Muhammad/AppData/Local/Programs/Python/Python310/python.exe  
c:/Users/Muhammad/Desktop/M.Sc\Thesis\soft/conventional.py  
Performance Metrics:  
Original File Size: 1020 bytes  
Encrypted File Size: 685.875 bytes  
AES Encryption Time: 0.031245 seconds  
RSA Encryption Time: 0.094528 seconds  
Huffman Encoding Time: 0.015622 seconds  
PS C:\Users\Muhammad\Desktop\M.Sc\Thesis\soft>
```

Figure 8: 1 kilobyte Encrypted. File encryption-compression time and space report for Conventional AES RES and Huffman.

Table 1: Encryption (Space) Complexity

SN	Size	AES-RSA Huffman(bytes)	Enhanced AES RSA-SHA- 256 Huffman (bytes)
1.	1024	686.125	724.625
2.	2048	1379.75	1445.375
3.	5120	3456.25	3627.125
4.	10240	6897.375	7253.125
5.	20480	13796.875	14511.5
6.	51200	34442.125	36275.5
7.	102400	68900.0	72558.0
8.	204800	137888.75	145110.95
9.	512000	344597.75	362782.875
10.	1055670	710337.75	747964.875

Table 2: Encryption Time Complexity

SN	Size on Disk	AES-RSA Huffman(seconds)	Enhanced AES RSA-SHA- 256 Huffman (seconds)
1.	1024	0.149950	0.001003
2.	2048	0.290410	0.216592
3.	5120	0.757891	0.789130
4.	10240	1.468064	1.764096
5.	20480	2.716959	2.969311
6.	51200	7.007834	7.244300
7.	102400	13.737787	14.286484
8.	204800	28.177013	27.328622
9.	512000	72.648272	72.406702
10.	1055670	141.920079	147.832119

Table 3: Security Strength (Avalanche Effect)

SN	Size on Disk	AES RSA Huffman Encoding	Modified AES RSA SHA- 256 Huffman Encoding
1.	1024	0.80886	0.8245
2.	2048	0.79454	0.8187
3.	5120	0.78789	0.8065
4.	10240	0.78005	0.7965
5.	20480	0.72988	0.7856
6.	51200	0.71688	0.7755
7.	102400	0.697546	0.7697
8.	204800	0.691254	0.7531
9.	512000	0.68755	0.7478
10.	1055670	0.68440	0.7354

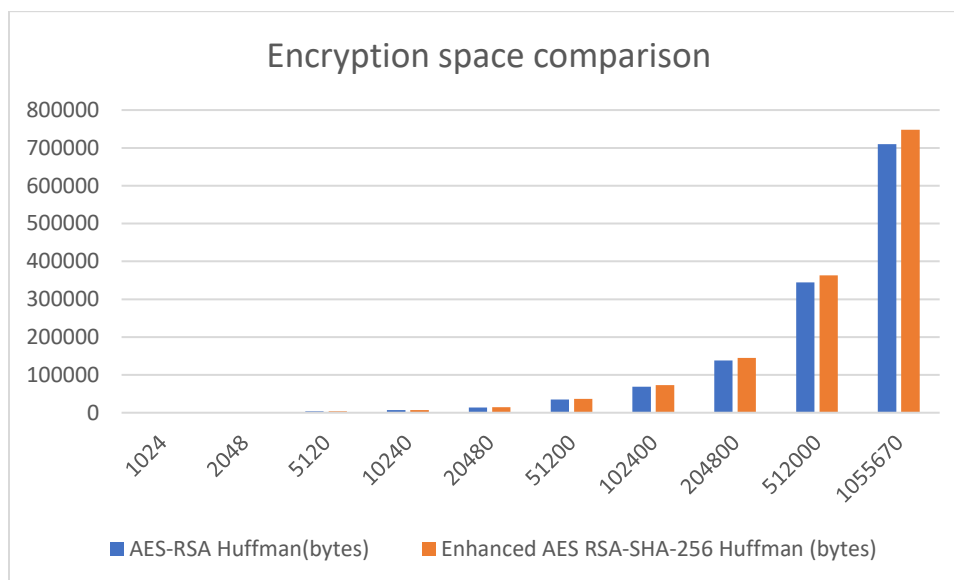


Figure 9: Data Encryption Space Comparison

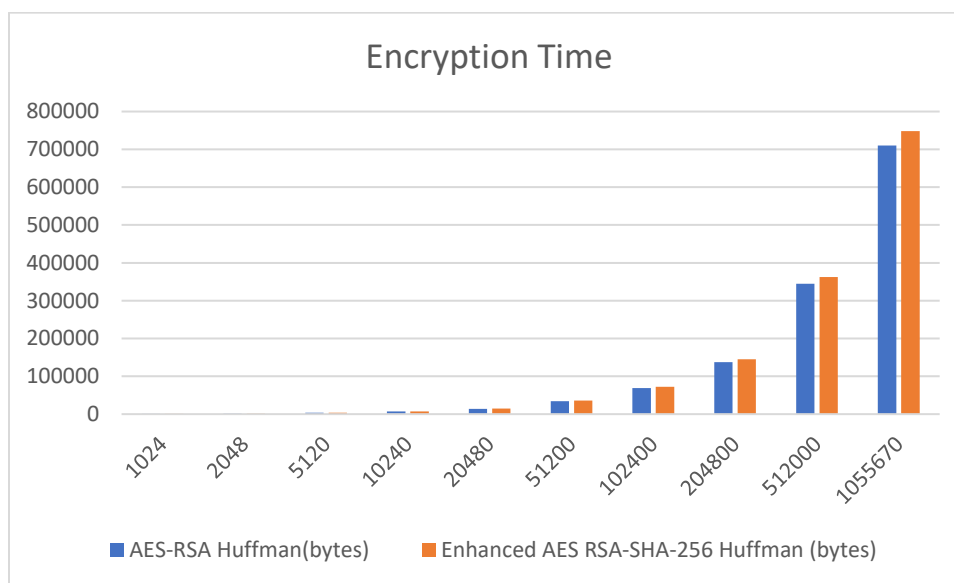


Figure 10: Data Encryption Time Comparison

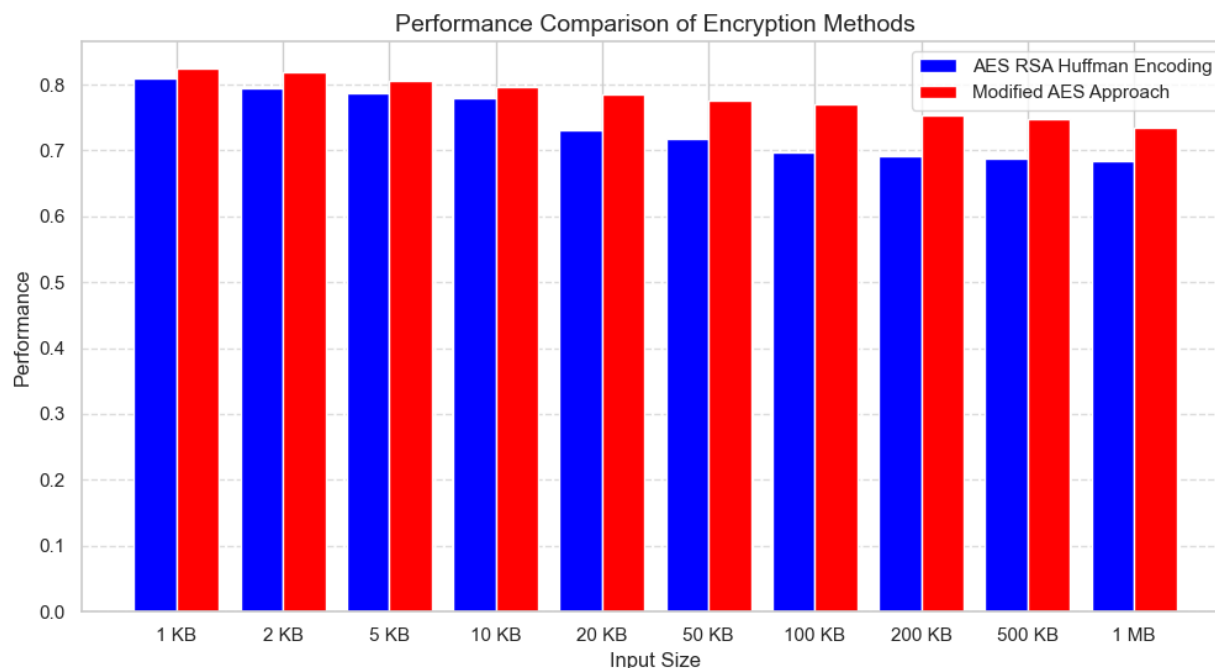


Figure 11: Data Encryption Security Strength Comparison

Table 4: Testing Machine

SN	Description	Value
1.	Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz
2.	Installed RAM	16.0 GB (15.9 GB usable)
3.	CPU Cores	4 Logical Processor(s)
4.	Local Disk	SSD 1TB
5.	Local Disk Location	Bus Number 1, Target Id 0, LUN 0
6.	OS Name	Microsoft Windows 11 Pro
7.	Version	10.0.22621 Build 22621
8.	System Manufacturer	HP
9.	SMBIOS Version	2.8
10.	BIOS Mode	Lagacy
11.	Base-Board Version	83.14
12.	System Type	64-bit operating system, x64-based processor

RESULT DISCUSSION

Figure 4 represents the authentication interface where the user enters their preferred password for validation. This stage is crucial as it serves as the entry point for securing data within the system. The authentication mechanism ensures that only authorized users can access encryption and decryption functions, reinforcing the confidentiality of sensitive information. Given that the system integrates AES and RSA encryption, the password may also play a role in key derivation and management ensuring a secure foundation for subsequent encryption processes.

The authentication page's design is a fundamental aspect of cybersecurity as weak authentication mechanisms can compromise the entire encryption framework. The system's ability to validate user credentials securely without exposing sensitive data is crucial. If multi-factor authentication (MFA) or additional security layers were incorporated the system's resilience against attacks such as quantum attempts and credential stuffing would significantly improve. The strength of the authentication phase ultimately dictates the security of the encryption process that follows.

Figure 5 displays the result of encrypting the user's password using the Advanced Encryption Standard (AES). AES is a widely used symmetric encryption algorithm known for its heftiness and efficiency. The encryption of the password before any further processing ensures that the plaintext credentials are not stored or transmitted in an unprotected form, mitigating risks associated with password leaks. Given that AES operates in different modes (e.g., ECB, CBC, GCM), the chosen mode impacts both security and performance.

Figure 6 showcases the AES key encrypted using the modified RSA algorithm, incorporating SHA-256. In this approach, instead of encrypting the AES key directly, SHA-256 is first applied to generate a hash, which is then encrypted using RSA. This method enhances security by reducing direct exposure of the AES key in its raw form, adding a cryptographic layer that ensures integrity and resistance to key-recovery attacks. Since RSA is traditionally vulnerable to certain attacks when encrypting small keys the use of SHA-256 addresses these concerns by increasing randomness in the ciphertext.

The integration of SHA-256 with RSA encryption enhances security by preventing attackers from easily reconstructing the original AES key, even if partial information about the encrypted key is exposed. The approach also introduces a challenge regarding key management, as decrypting the AES key requires both the RSA private key and knowledge of the hashing mechanism. While this modification adds an additional processing step, the security benefits outweigh the computational overhead, making it a viable improvement over conventional RSA encryption methods.

Figure 7 depicts the application of Huffman encoding, a lossless data compression technique. Huffman encoding optimizes storage efficiency by assigning shorter binary codes to more frequently occurring characters in the data. This step is particularly useful in an encryption system because it reduces the size of encrypted data before transmission or storage, thereby improving overall efficiency. By incorporating Huffman encoding, the system achieves better space utilization without compromising data integrity or security.

In this encryption framework, Huffman encoding complements AES and RSA by ensuring that encrypted data occupies minimal storage space while maintaining recoverability. Since Huffman

encoding is reversible, the original data can be reconstructed without loss after decryption. However, the effectiveness of this approach depends on the entropy of the input data—highly random data may not compress significantly. Nevertheless, this step provides additional optimization, making the hybrid system more efficient in handling encrypted files.

Figure 8 presents an analysis of encryption and compression times for the conventional AES-RSA-Huffman system versus the modified AES-RSA-SHA-256-Huffman system. The report measures the efficiency of each method in terms of computational time and space utilization. The results indicate that while the modified system introduces slight increases in processing time due to additional hashing operations, it provides a more secure encryption framework. The trade-off between security and computational efficiency is a key consideration in encryption system design. The findings suggest that the enhanced system remains practical for real-world applications, as the increased encryption time does not significantly impact usability. The space utilization report also demonstrates that while Huffman encoding contributes to compression, the added cryptographic steps slightly increase the final storage requirements. Overall, the modifications provide stronger security with a manageable increase in computational overhead, making the system a viable enhancement over conventional methods.

Table 1 compares the space complexity of the conventional AES-RSA-Huffman system and the modified AES-RSA-SHA-256-Huffman system across different file sizes. The results show that while both systems maintain reasonable storage footprints, the modified system requires slightly more space due to the additional SHA-256 hashing step. For instance, for a 1,024-byte file, the conventional system requires 686.125 bytes, while the modified system uses 724.625 bytes. This pattern is consistent across all file sizes, with the space requirement increasing proportionally.

The additional storage overhead is a result of the SHA-256 hashing step, which expands the ciphertext before RSA encryption. Although this increases the storage requirement slightly, the trade-off for improved security is justifiable. The increase in space is not exponential, meaning that for most practical applications, the modified system remains an efficient choice. The slight increase in space usage ensures better protection against cryptographic attacks while maintaining reasonable storage efficiency.

Table 2 evaluates the encryption time complexity of the two encryption models. The results indicate that for smaller file sizes (e.g., 1,024 bytes), the modified system encrypts data in less than a millisecond, showcasing negligible performance impact. However, as file sizes increase, the additional hashing step causes a slight increase in encryption time. For instance, a 512 KB file is encrypted in 72.40 seconds using the modified system, compared to 72.64 seconds for the conventional system.

The results demonstrate that while the enhanced system has a slight time overhead, the increase is not substantial enough to hinder performance. The hashing operation adds a predictable delay, but its impact diminishes at larger file sizes due to parallel processing and optimization in modern processors. This finding suggests that the modified approach is practical for real-world applications where security is a priority, as the encryption delay is not excessive.

Table 3 measures the security strength of both encryption models using the avalanche effect an indicator of how much the ciphertext changes when a single bit in the plaintext is altered. The results show that the modified AES-RSA-SHA-256-Huffman system has a higher avalanche effect

across all file sizes. For instance, at 1,024 bytes, the conventional system records an avalanche effect of 0.80886, while the modified system achieves 0.8245. This trend continues across larger files, with the modified system consistently demonstrating superior security.

A higher avalanche effect indicates stronger diffusion properties, meaning small changes in input data result in significantly different ciphertexts. This enhances resistance against differential cryptanalysis attacks, making the modified system more stronger. The improvement in security strength justifies the minor increases in encryption time and space complexity as the system provides a stronger defense against cryptographic attacks while maintaining reasonable performance.

Figures 9, 10, and 11 illustrate the space, time, and security comparisons between the conventional and modified encryption systems. These visual comparisons reinforce the tabular findings, demonstrating the trade-offs between security and performance. Figure 4.6 highlights the slight increase in storage space required by the modified system. Figure 4.7 showcases the minimal increase in encryption time, while Figure 4.8 confirms the improved security strength.

Table 4 provides the specifications of the testing environment, which is crucial for evaluating encryption performance. The system is equipped with an Intel Core i5 processor, 16GB RAM, and an SSD, ensuring that encryption operations are tested on a capable platform. These specifications suggest that the results may vary on lower-end devices, but the overall trend of efficiency versus security should remain consistent.

CONCLUSION

The findings of this research confirm that the combination of modified RSA with SHA-256, AES encryption, and Huffman encoding provides an effective security framework for banking transactions. The study demonstrates that integrating SHA-256 into RSA encryption enhances the protection of AES keys, making brute-force and cryptanalytic attacks more difficult. Although the space and computational time increased marginally compared to conventional models, the security improvement outweighs the added complexity.

Furthermore, Huffman encoding effectively compresses encrypted transaction data, optimizing storage and transmission efficiency. This ensures that banking institutions can secure sensitive data while reducing processing time and costs associated with data transmission and storage. The experimental results validate that the proposed hybrid model achieves a balance between security and performance, making it suitable for real-world applications where security, speed, and storage efficiency are critical.

RECOMMENDATION

Based on the study findings, the following recommendations are made:

1. **Adoption in Banking Systems** – Financial institutions should consider integrating the proposed hybrid encryption-compression model into their security architecture to enhance the protection of sensitive banking transactions.

2. **Further Performance Optimization** – Future research should focus on optimizing the computational efficiency of the RSA-SHA-256 encryption process to minimize the slight increase in encryption time.
3. **Implementation in Real-World Transactions** – The hybrid model should be tested in live banking environments to assess its effectiveness under real-time transaction loads and varying network conditions.
4. **Integration with Emerging Technologies** – The model should be explored for use in blockchain-based banking transactions and IoT-enabled financial systems to enhance security in decentralized financial networks.
5. **Multi-Factor Authentication Enhancement** – The encryption model should be combined with biometric authentication methods to create a more robust security framework for banking applications.

REFERENCES

- Abhilash, A., Shenoy, S. S., & Shetty, D. K. (2023). Overview of Corporate Governance Research in India: A Bibliometric Analysis. *Cogent Business & Management*, 10(1), 2182361. <https://doi.org/10.1080/23311975.2023.2182361>
- Agur, I., Peria, S. M., & Rochon, C. (2020). Digital financial services and the pandemic: Opportunities and risks for emerging and developing economies. *International Monetary Fund Special Series on COVID-19, Transactions*, 1, 2–1. https://www.imf.org/~media/Files/Publications/covid19-special-notes/en-special-series-on-covid-19-digital-financial-services-and-the-pandemic.ashx?la=en&utm_medium=email&utm_source=govdelivery
- Ajagbe, S. A., Adeniji, O. D., Olayiwola, A. A., & Abiona, S. F. (2024). Advanced Encryption Standard (AES)-Based Text Encryption for Near Field Communication (NFC) Using Huffman Compression. *SN Computer Science*, 5(1), 156. <https://doi.org/10.1007/s42979-023-02486-6>
- Alenizi, A., Mohammadi, M. S., Al-Hajji, A. A., & Ansari, A. S. (2024). A Review of Image Steganography Based on Multiple Hashing Algorithm. *Computers, Materials & Continua*, 80(2). https://www.researchgate.net/profile/Arshiya-Ansari-2/publication/382661222_A_Review_of_Image_Steganography_Based_on_Multiple_Hashing_Algorithm/links/66c0d7db145f4d355361f107/A-Review-of-Image-Steganography-Based-on-Multiple-Hashing-Algorithm.pdf
- Altigani, A., Hasan, S., Barry, B., Naserelden, S., Elsadig, M. A., & Elshoush, H. T. (2021). A polymorphic advanced encryption standard—a novel approach. *IEEE Access*, 9, 20191–20207. <https://ieeexplore.ieee.org/abstract/document/9321317/>
- Andersson, M. (2023). *Optimizing the computation of password hashes*. <https://helda.helsinki.fi/server/api/core/bitstreams/23a37f74-a162-4473-b894-5da77f0627d1/content>

- Ashila, M. R., Atikah, N., Rachmawanto, E. H., & Sari, C. A. (2019). Hybrid AES-Huffman coding for secure lossless transmission. *2019 Fourth International Conference on Informatics and Computing (ICIC)*, 1–5. <https://ieeexplore.ieee.org/abstract/document/8985899/>
- Erdal, E., & Ergüzen, A. (2019). An efficient encoding algorithm using local path on huffman encoding algorithm for compression. *Applied Sciences*, 9(4), 782. <https://www.mdpi.com/2076-3417/9/4/782>
- Gajjala, R. R., Banchhor, S., Abdelmoniem, A. M., Dutta, A., Canini, M., & Kalnis, P. (2020). Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning. *Proceedings of the 1st Workshop on Distributed Machine Learning*, 21–27. <https://doi.org/10.1145/3426745.3431334>
- Grassi, L., Leander, G., Rechberger, C., Tezcan, C., & Wiemer, F. (2021). Weak-Key Distinguishers for AES. In O. Dunkelman, M. J. Jacobson, & C. O’Flynn (Eds.), *Selected Areas in Cryptography* (Vol. 12804, pp. 141–170). Springer International Publishing. https://doi.org/10.1007/978-3-030-81652-0_6
- Habib, A., Islam, M. J., & Rahman, M. S. (2020). A dictionary-based text compression technique using quaternary code. *Iran Journal of Computer Science*, 3(3), 127–136. <https://doi.org/10.1007/s42044-019-00047-w>
- Haldar-Iversen, S. (2020). *Improving the text compression ratio for ASCII text Using a combination of dictionary coding, ASCII compression, and Huffman coding* [Master’s Thesis, UiT Norges arktiske universitet]. <https://munin.uit.no/handle/10037/20517>
- Haryaman, A., Amrita, N. D. A., & Redjeki, F. (2024). SECURE AND INCLUSIVE UTILIZATION OF SHARED DATA POTENTIAL WITH MULTI-KEY HOMOMORPHIC ENCRYPTION IN BANKING INDUSTRY. *Journal of Economics, Accounting, Business, Management, Engineering and Society*, 1(9), 1–13. <http://kisainstitute.com/index.php/kisainstitute/article/view/36>
- Herzog, C., Tong, V. V. T., Wilke, P., van Straaten, A., & Lanet, J.-L. (2020). Evasive Windows Malware: Impact on Antiviruses and Possible Countermeasures. *Proceedings of the 17th International Joint Conference on E-Business and Telecommunications*, 302–309. <https://doi.org/10.5220/0009816703020309>
- Islam, M., Nurain, N., Kaykobad, M., Chellappan, S., & Islam, A. B. M. A. A. (2019). HELiOS: Huffman coding based lightweight encryption scheme for data transmission. *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 70–79. <https://doi.org/10.1145/3360774.3360829>
- Javaid, M., Haleem, A., Singh, R. P., Suman, R., & Khan, S. (2022). A review of Blockchain Technology applications for financial services. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 2(3), 100073. <https://www.sciencedirect.com/science/article/pii/S2772485922000606>
- Kaffah, F. M., Gerhana, Y. A., Huda, I. M., Rahman, A., Manaf, K., & Subaeki, B. (2020). E-mail message encryption using Advanced Encryption Standard (AES) and Huffman compression engineering. *2020 6th International Conference on Wireless and Telematics (ICWT)*, 1–6. <https://ieeexplore.ieee.org/abstract/document/9243651/>

- Kaur, P., Kaur, R., Kaur, A., & Sharma, V. K. (2023). Privacy Preservation and Secured Data Storage on Cloud Using Encryption Algorithms. *2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*, 1374–1378. <https://ieeexplore.ieee.org/abstract/document/10465702/>
- Kishor Kumar, R., Yogesh, M. H., Raghavendra Prasad, K., Sharankumar, & Sabareesh, S. (2024). 256-Bit AES Encryption Using SubBytes Blocks Optimisation. In V. K. Gunjan, A. Kumar, J. M. Zurada, & S. N. Singh (Eds.), *Computational Intelligence in Machine Learning* (Vol. 1106, pp. 621–628). Springer Nature Singapore. https://doi.org/10.1007/978-981-99-7954-7_56
- Kumar, M. R. R., Josna, B. A., Lawvanyaa, R., & Shruthi, S. (2023). *ADVANCED SECURITY USING ENCRYPTION, COMPRESSION AND STEGANOGRAPHY TECHNIQUES*. https://www.academia.edu/download/104182962/IRJET_V10I603.pdf
- Nidhi, M., & Kadam, S. A. (n.d.). *A STUDY OF ACADEMIC INSTITUTION'S DIGITAL CERTIFICATES PREFERENCES FOR WEBSITE SECURITY*. Retrieved October 25, 2024, from https://www.researchgate.net/profile/Sachin-Kadam-14/publication/362518466_A_STUDY_OF_ACADEMIC_INSTITUTION'S_DIGITAL_CERTIFICATES_PREFERENCES_FOR_WEBSITE_SECURITY/links/634ea41112cbac6a3ed72f91/A-STUDY-OF-ACADEMIC-INSTITUTIONS-DIGITAL-CERTIFICATES-PREFERENCES-FOR-WEBSITE-SECURITY.pdf
- Prasanna, R., Prathaban, B. P., Jenath, M., Rajendran, S., & Ashokkumar, M. (2024). Computational framework for human detection through improved ultra-wide band radar system. *International Journal for Multiscale Computational Engineering*, 22(1). <https://www.dl.begellhouse.com/journals/61fd1b191cf7e96f,6129d44f1682fc8e,1e068d4e5c88fe0e.html>
- Rahman, Md. A., & Hamada, M. (2023). A prediction-based lossless image compression procedure using dimension reduction and Huffman coding. *Multimedia Tools and Applications*, 82(3), 4081–4105. <https://doi.org/10.1007/s11042-022-13283-3>
- Reza, M. S., Riya, S. A., Alam, S. A., & Hossain, M. A. A. (2019). *Study on Text Compression* [PhD Thesis, United International University]. <http://dspace.uiu.ac.bd/handle/52243/822>
- Rivera, C., Di, S., Tian, J., Yu, X., Tao, D., & Cappello, F. (2022). Optimizing huffman decoding for error-bounded lossy compression on gpus. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 717–727. <https://ieeexplore.ieee.org/abstract/document/9820677/>
- SANDHU, S. (2021). *LOSSLESS DATA COMPRESSION: AN OVERVIEW*. <https://www.ubishops.ca/wp-content/uploads/sandhu20211029.pdf>
- Sari, C. A., Ardiansyah, G., & Rachmawanto, E. H. (2019). An improved security and message capacity using AES and Huffman coding on image steganography. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 17(5), 2400–2409. <http://telkomnika.uad.ac.id/index.php/TELKOMNIKA/article/view/9570>
- Sivanandam, L., Periyasamy, S., & Oorkavalan, U. M. (2020). Power transition X filling based selective Huffman encoding technique for test-data compression and Scan Power

- Reduction for SOCs. *Microprocessors and Microsystems*, 72, 102937. <https://www.sciencedirect.com/science/article/pii/S0141933119304399>
- Smid, M. E. (2021). Development of the advanced encryption standard. *Journal of Research of the National Institute of Standards and Technology*, 126. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9682931/>
- Tabassum, T., & Mahmood, M. A. (2020). A multi-layer data encryption and decryption mechanism employing cryptography and steganography. *2020 Emerging Technology in Computing, Communication and Electronics (ETCCE)*, 1–6. <https://ieeexplore.ieee.org/abstract/document/9350908/>
- Taneja, A., & Shukla, R. K. (2021). Comparative Study of RSA with Optimized RSA to Enhance Security. In A. Kumar & S. Mozar (Eds.), *ICCCE 2020* (Vol. 698, pp. 975–996). Springer Nature Singapore. https://doi.org/10.1007/978-981-15-7961-5_91
- Wahab, O. F. A., Khalaf, A. A., Hussein, A. I., & Hamed, H. F. (2021). Hiding data using efficient combination of RSA cryptography, and compression steganography techniques. *IEEE Access*, 9, 31805–31815. <https://ieeexplore.ieee.org/abstract/document/9356603/>
- Yusuf, A. Y., Gambo, F. L., Shin, H., & Miyim, A. M. (2023). Hybrid Encryption of ElGamal-AES with Huffman Coding for Efficient Data Communication. *The Journal of Contents Computing*, 5(2), 685–697. <https://www.dbpia.co.kr/Journal/articleDetail?nodeId=NODE11660967>