

The HAJ: Initial Steps of a Novel Cryptographic Algorithm

Daniel Alarcón-Narváez¹, Fausto A. Jacques-García², L.A. Lizama-Pérez^{2,*},
S.L. Canchola-Magdaleno¹, R.C. Chaparro¹

¹ Autonomous University of Queretaro, Computer Science, Querétaro,
Mexico

² Universidad Autónoma Metropolitana, Basic Sciences and Engineering,
Mexico

*l.lizama@correo.1er.uam.mx, daniel.alarcon@uaq.mx

Abstract. We present the Hill-Alarcón-Jacques (HAJ) algorithm, a symmetric cryptographic construction that combines the classical Hill cipher with the Gauss-Jacques method for modular matrix inversion. The principal novelty of this initial HAJ study is the integration of a matrix-based encryption/decryption process with a modular elimination procedure for obtaining the inverse matrices required during decryption. Particular attention was paid to the use of large matrix dimensions, as matrix size directly influenced the computational cost of the proposed approach. The construction incorporated multi-round encryption, block-based processing, dynamic subkey generation, and flexible matrix dimensions. The experimental evaluation considered both full-size matrices and block-based configurations, using national-anthem plaintexts and thirty runs per configuration across desktop, macOS, and MATLAB Online environments. The results showed comparatively stable memory consumption for a given configuration, whereas execution time varied according to plaintext length, matrix dimension, block size, and computational platform. These experiments provided an initial empirical characterization of HAJ and established a computational baseline for analyzing its scalability and resource requirements. The results represented the initial algorithmic and experimental stage of a broader investigation into matrix-based cryptographic primitives, while also identifying cryptanalytic and theoretical questions that remained to be addressed before stronger security claims could be established.

Keywords: Gauss-Jacques method, Hill cipher, modular inverse calculation, security, symmetric cryptography, matrix-based cryptography, lightweight cryptography.

El HAJ: Primeros pasos de un nuevo algoritmo criptográfico

Resumen. Presentamos el algoritmo Hill-Alarcón-Jacques (HAJ), una construcción criptográfica simétrica que combina el cifrado clásico de Hill

con el método de Gauss-Jacques para la inversión de matrices modulares. La principal novedad de este estudio inicial sobre HAJ radica en la integración de un proceso de cifrado y descifrado basado en matrices con un procedimiento de eliminación modular para obtener las matrices inversas requeridas durante el descifrado. Se prestó especial atención al uso de matrices de grandes dimensiones, dado que el tamaño de la matriz influía directamente en el coste computacional del enfoque propuesto. La construcción incorporó cifrado de múltiples rondas, procesamiento por bloques, generación dinámica de subclaves y dimensiones matriciales flexibles. La evaluación experimental consideró tanto matrices de tamaño completo como configuraciones basadas en bloques, utilizando textos en claro consistentes en himnos nacionales y realizando treinta ejecuciones por configuración en entornos de escritorio, macOS y MATLAB Online. Los resultados mostraron un consumo de memoria comparativamente estable para una configuración dada, mientras que el tiempo de ejecución varió en función de la longitud del texto en claro, la dimensión de la matriz, el tamaño del bloque y la plataforma computacional. Estos experimentos proporcionaron una caracterización empírica inicial de HAJ y establecieron una línea base computacional para analizar su escalabilidad y sus requisitos de recursos. Los resultados representaron la etapa algorítmica y experimental inicial de una investigación más amplia sobre primitivas criptográficas basadas en matrices, al tiempo que identificaron cuestiones criptoanalíticas y teóricas pendientes de abordar antes de poder establecer afirmaciones de seguridad más sólidas.

Palabras clave: Método de Gauss-Jacques, cifrado de Hill, cálculo de la inversa modular, seguridad, criptografía simétrica, criptografía basada en matrices, criptografía ligera.

1 Introduction

Cryptography is the study and practice of techniques for securing communication in the presence of unauthorized third parties. In modern cryptography, there are two main types of encryption: symmetric cryptography and asymmetric cryptography [1].

Symmetric cryptography, also known as secret-key cryptography, uses a shared key between the sender and receiver to encrypt and decrypt the message. The security of this method depends on the shared key being known only to the sender and receiver. Examples of symmetric cryptography algorithms are DES (Data Encryption Standard), AES (Advanced Encryption Standard), and the Hill-Cipher symmetric algorithm [2].

On the other hand, asymmetric cryptography, also known as public-key cryptography, uses a pair of keys: a public key and a private key. The public key is shared with anyone who wants to send a message to the owner of the private key. The owner uses their private key to decrypt the message. Examples of asymmetric cryptography algorithms are RSA (Rivest-Shamir-Adleman), Diffie-Hellman, and ECC (Elliptic Curve Cryptography) [2].

Both types of encryption have their advantages and disadvantages and are used in different situations. The choice of the appropriate encryption method depends on the nature of the information to be encrypted and the security needs of the system.

In recent years, the design of lightweight cryptographic algorithms has gained considerable attention, especially for Internet of Things (IoT) and resource-constrained environments. Comprehensive evaluations of symmetric ciphers for IoT show that algorithmic efficiency and robustness are equally critical, as trade-offs between security and performance determine practical deployment [3,4]. These works highlight the need for symmetric ciphers that combine mathematical rigor with computational feasibility.

One of the main challenges in Hill-type ciphers is their vulnerability to linear cryptanalysis and known-plaintext attacks. Several modern studies have focused on analyzing and breaking variants of the Hill cipher. For example, Wen et al. [5] demonstrated that Hill-based schemes combined with chaotic systems can still be attacked using statistical analysis. Similarly, Teseleanu [6] analyzed a Hill-plus-chaos variant and showed weaknesses when subjected to standard cryptanalytic techniques. These findings suggest that any novel Hill variant must be thoroughly evaluated against both classical and modern attacks.

At the same time, mathematical advances in modular inverse computation remain relevant for cryptographic constructions. Bufalo [7] provided a detailed analysis of modular inverse properties and highlighted their applications in cryptography. Recent developments also include efficient methods for inversion modulo powers of integers, such as those presented by Xu et al. [8], which illustrate the ongoing interest in optimizing matrix inversion techniques for cryptographic use cases.

Finally, new designs continue to explore robust Hill-based algorithms. Rrgh-out et al. [9] proposed an image encryption algorithm based on a novel Hill variant, showing the community's continuous exploration of this family of ciphers.

However, the existing literature indicates that security analysis—such as resistance to differential and linear attacks, as well as statistical tests of randomness—is often missing or insufficient, which motivates the present work. In this context, we introduce a new symmetric cryptographic algorithm called Hill-Alarcon-Jacques (HAJ), which, to the best of our knowledge, represents the first attempt to combine the Hill cipher with the Gauss-Jacques method for modular inverse calculation. This novel integration has not been explored in prior works, making HAJ a unique and previously unreported contribution to the field of symmetric cryptography.

The remainder of this paper is organized as follows. Section 2 describes the computational environment. Section 3 introduces the theoretical background, including the classical Hill cipher and the Gauss-Jacques modular inverse method. Section 4 details the methodology of the proposed Hill-Alarcón-Jacques (HAJ) algorithm. Section 5 presents the algorithm pseudocode. Section 6 reports both

performance results. Section 7 concludes the paper and discusses future research directions.

2 Materials

1. Equipment used:
 - Desktop HP computer with the following specifications. Processor: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz 2.90 Ghz RAM: 8 GB Operating System: Windows 11 Home.
2. Equipment used:
 - Desktop Macbook Air computer with the following specifications: Processor: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz 2.90 Ghz, RAM: 8 GB, Operating System: macOS Monterrey.
3. Software used:
 - MATLAB (Online Trial Version): The online version of MATLAB provided by MathWorks was utilized for performing numerical computations, implementing algorithms, and visualizing results. It was accessed through a compatible web browser, and the programming and data analysis capabilities of the MATLAB environment were leveraged.
 - MATLAB version R2021a (Free Desktop Version): In addition to the online version, the free desktop version of MATLAB was also employed. Specifically, version R2021a was installed on the local computer and used for code development, debugging, conducting experiments, and performing additional analysis.

3 Theoretical Framework

Our proposed algorithm follows the general principles of the Hill cipher. It involves dividing the plaintext into blocks, each of which is represented as a matrix. These matrices are then multiplied by a secret key matrix, resulting in the ciphertext.

The key innovation lies in the calculation of the matrix inverse using the Gauss-Jacques method, which offers a more efficient and modular approach compared to traditional methods. By utilizing modular inverse calculation, our algorithm ensures the encryption process remains secure even with a relatively small key size. All the theoretical information is found in the article Towards a Symmetric Crypto Algorithm: The HAJ [11]. The proposed HAJ algorithm, while focused on enhancing cryptographic robustness, also has the potential to be adapted for lightweight environments such as the Internet of Things (IoT), due to its modular structure and efficient matrix-based operations. In resource-constrained environments, encryption algorithms must ensure both computational efficiency and cryptographic strength. A recent approach proposes a lightweight symmetric cryptosystem based on AES and a chaotic S-box, designed for embedded platforms. Using techniques like the Lorenz system and Hilbert scan patterns, the system achieved validated cryptographic properties through NIST tests and demonstrated efficient performance on real-world sensors [12].

3.1 Hill Cipher

The Hill-Cipher algorithm is a linear substitution encryption algorithm that uses a matrix as a key to encrypt and decrypt the message. It was proposed by Lester S. Hill in 1929 [13], and is based on the key matrix, which is an invertible square matrix. The Hill Cipher algorithm is used in data security applications to protect information from malicious attacks. This algorithm is considered one of the first symmetric encryption algorithms and has been used in military and government applications for decades. However, its use has decreased due to the arrival of more secure and efficient algorithms [14].

In the Hill Cipher, the original message is divided into blocks of length n , where n is the size of the key matrix. Each block is multiplied by the key matrix, and the result is reduced modulo a prime number to obtain the encrypted message. The decryption process involves multiplying the encrypted message by the inverse matrix of the key and reducing modulo the prime number.

Below are some relevant experiments and references related to the Hill Cipher:

Experiment 1: Known-plaintext attack In this experiment, an attacker knows some blocks of plaintext and the corresponding blocks of ciphertext. Using this information, the attacker can calculate the key matrix by using linear algebra and decrypt the rest of the ciphertext message [16].

Experiment 2: Known-ciphertext attack In this experiment, an attacker knows some blocks of ciphertext and the corresponding blocks of plaintext.

Using this information, the attacker can calculate the inverse matrix of the key by using linear algebra and decrypt the rest of the ciphertext message [18].

Experiment 3: Frequency analysis In this experiment, an attacker can use frequency analysis to decrypt the ciphertext message. The attacker can use the frequency of letters in the ciphertext message to determine the most common letters and, from there, deduce the key and decrypt the entire message [16].

Experiment 4: Side-channel attack Side-channel leakages, such as electromagnetic emissions, can significantly reduce the security margin of symmetric encryption algorithms. Recent work proposed a key classification framework based on deep transfer learning, using a divide-and-conquer strategy over semi-byte structures to reduce key enumeration complexity. The model, composed of symmetric subnetworks and an end-to-end mapping design, achieved up to 91% attack accuracy while maintaining robustness under small-sample conditions [3].

Although the Hill Cipher is a solid encryption algorithm, it is not invulnerable to attacks. The experiments mentioned above demonstrate some of the vulnerabilities of the Hill Cipher and the importance of choosing an appropriate key matrix and ensuring that the prime number used for modulo reduction is large enough. Additionally, it is important to remember that encryption algorithms are one of the many tools used in data security, and data security depends on a combination of technical security measures and management procedures.

3.2 Gauss-Jacques Method for Modular Inverse Calculation

The Gauss-Jacques method plays a crucial role in our proposed symmetric cryptographic algorithm for modular inverse calculation. This method allows us to efficiently compute the modular inverse of variable-sized matrices, making it particularly suitable for cryptographic applications.

The Gauss-Jacques method is implemented as a separate function, taking two inputs: the matrix 'K' and the prime number 'm'. It ensures that 'K' is a square matrix, and if not, it raises an error. The function returns two outputs: the modular inverse matrix 'InvMod' and the identity matrix 'I'.

The algorithm proceeds as follows:

- The original matrix 'K' is augmented with the identity matrix to form an augmented matrix.
- For each row in 'K', a pivot element is identified. If the pivot is zero, a row interchange operation is performed to ensure a non-zero pivot.
- The extended Euclidean algorithm is used to calculate the modular inverse 'x' of the pivot element with respect to 'm'.
- The current row of 'K' is multiplied by 'x' modulo 'm' to obtain the updated row.
- For each row other than the current row, row operations are performed to eliminate the non-zero elements below and above the pivot element.
- Finally, the resulting matrix is divided into two parts: 'InvMod', which contains the modular inverse matrix, and 'I', which represents the identity matrix.

3.3 Extended Euclidean Algorithm

The extended Euclidean algorithm is a fundamental tool for finding the greatest common divisor (gcd) of two integer numbers in a Euclidean space.

It is used within the Gauss-Jacques method to calculate the modular inverse of the pivot element.

The extended Euclidean algorithm function takes two positive integers, 'a' and 'b', as inputs. It returns three integers, 'x', 'y', and 'd', such that 'd' represents the gcd of 'a' and 'b', and $ax + by = d$.

The algorithm proceeds as follows:

- If 'b' is equal to zero, the function sets 'x' to 1, 'y' to 0, and 'd' to 'a' before returning.
- Otherwise, the function recursively calls itself with 'b' and 'mod(a,b)' as the new inputs, storing the results in 'x1', 'y1', and 'd1'.
- 'x' is set to 'y1', 'y' is set to 'x1 - floor(a/b)*y1', and 'd' is set to 'd1'.

4 Methodology

The HAJ algorithm retains the Hill cipher's block-matrix multiplication as the core encryption operation. The novelty lies in the computation of the modular

inverse of each key matrix: instead of the standard adjugate/determinant approach, HAJ applies the Gauss-Jacques elimination method to obtain the inverse modulo m . During each round, a fresh key matrix is generated and its inverse computed by Gauss-Jacques, enabling efficient decryption and supporting the multi-round, subkey structure.

4.1 A Round of HAJ

The previous section describes the generalization for creating and understanding how the HAJ algorithm would work, i.e., the summation of the blocks and the 7 times it will be performed. Now, in the following paragraphs, we will describe how we intend to generate each block, the bits that will be used, the size of each block, and how the primary and secondary keys will be generated.

Obtaining the String Size There are different functions to know the size of a string, where any of them will be used to find that value. What is interesting in this section is that we seek for that value to be a multiple of three hundred. Therefore, we will use the modulo 300 of the string length to verify it, and if necessary, some characters will be added to the string to make it a multiple of three hundred.

Example code:

```
if s % 300 != 0
    a = 300 - s;
    for i = 1:a
        p(lp+i) = 'Q';
    end
end
```

Key Generation The primary key will be generated using the Gauss-Jacques method, where the size of the matrix will be determined by the size of the resulting block in the previous steps. However, for each block, a primary key will be generated, and for each round of the 7 rounds, secondary or sub-keys will be generated to improve security at each iteration.

For the secondary keys, we will use the left circular shift. A circular shift in this case to the left is the operation of rearranging the entries in a string, either by moving the final entry to the first position while shifting all other entries to the next position, or by performing the inverse operation. A circular shift is a special type of cyclic permutation, which in turn is a special type of permutation.

This operation is used in various fields and different algorithms. An example of this in cryptography is the Data Encryption Standard algorithm, where positions are moved from the order of the primary key with respect to a shift list determined beforehand.

4.2 Encryption and Decryption Process

To demonstrate the functionality of our proposed symmetric cryptographic algorithm, we implemented the encryption and decryption processes using the provided code. The algorithm utilizes the Hill cipher with modular inverse calculation using the Gauss-Jacques method.

For encryption, the algorithm requires the selection of a prime number, denoted as 'm', greater than the highest ASCII value intended for encryption. Additionally, the number of matrices, 'num_matrices', and the block size, 'block_size', need to be defined. The matrices are generated randomly, ensuring that the determinant is non-zero modulo 'm'. This approach aligns with recent advancements in symmetric key generation that integrate hybrid models based on Latin square matrices and subtractive random number generators to enhance key randomness, diffusion, and resistance to brute-force attacks [2].

The plaintext is obtained from the user and is processed in blocks. If the length of the plaintext is not a multiple of the block size, padding is applied to make it a multiple. The encryption process involves multiplying the block of plaintext with the corresponding key matrix, ensuring modulo 'm' arithmetic.

The decryption follows a similar process but uses the inverse of the key matrix calculated using the Gauss-Jacques method implemented in MATLAB [20]. The encrypted text is processed in blocks, and the decryption is performed by multiplying each encrypted text block with the inverse matrix modulo 'm'.

The resulting decrypted plaintext is displayed as the final output of the algorithm.

The algorithm 4.3 shows the pseudocode for a round of HAJ where the K is the same size as plaintext. For example, if the plaintext is about 170 characters, then the K is going to be 170x170 size and the algorithm 5 shows the pseudocode by blocks.

4.3 Pseudocode of the HAJ Round

In this section, we present a pseudocode of the HAJ Round algorithm.

Listing 1.1. Pseudocode for text encryption and decryption:

```
m = 257;
seeds = [36, 33, 88, 91, 1];

File = 'NationalAnthems/NameAnthem.txt';

fid = fopen(File, 'rb');
p = fread(fid, 'uint8');
fclose(fid);

lonp = length(p);

r = mod(lonp, 3);
```

```

if  $r \sim = 0$ 
    pad = 3 - r;
    p = [p ; zeros(pad,1)];
end

results = zeros(30,4);

for i = 1:length(seeds)

    seed = seeds(i);
    rng(seed);

    n = 3;
    k = randi([0 m-1], n, n);

    c = mod(k * reshape(p(1:3),3,1), m);

    InvMod = modular_inverse(k, m);
    p_decrypted = mod(InvMod * c, m);

end

writematrix(results, 'results.csv');

```

5 The HAJ: Pseudocode

In this section, we present pseudocode of the HAJ algorithm.

Listing 1.2. Pseudocode for text encryption and decryption with multiple rounds:

```

m = 257 # Modulus for encryption
seeds = [280, 494, 50, ..., 404, 181] # List of 30 seeds
File = 'NationalAnthems/NameAnthem.txt' # Path to the text file

# Read the text file
file = open(File, 'rb')
p = file.read()
file.close()

# Ensure the length of p is a multiple of 300
if len(p) % 300 != 0:
    # pad p with zeros

# Prepare the results array
results = [[0 for _ in range(4)] for _ in range(len(seeds))]

```

```
numbloque = 3
tamanho_bloque = len(p) // numbloque

for j in range(len(seeds)):
    # Initialize random seed
    seed = generate_random_seed()
    initialize_random_number_generator(seed)

    for ronda in range(num_rondas):
        for i in range(numbloque):
            k = generate_random_matrix(tamanho_bloque, tamanho_bloque)

            encryption_file = open('encryption_results.txt', 'w')
            decryption_file = open('decryption_results.txt', 'w')

            # Encrypt block by block, 7 times
            c = []
            for ronda in range(num_rondas):
                for i in range(0, len(p), tamanho_bloque):
                    block = get_current_block(p, i, tamanho_bloque)
                    encrypted_block = encrypt_block(block)
                    c.append(encrypted_block)
                write_encryption_results_to_file(encryption_file, c)

            # Decrypt block by block, 7 times
            p_decifrado = []
            for ronda in range(num_rondas):
                for i in range(0, len(c), tamanho_bloque):
                    block = get_current_block(c, i, tamanho_bloque)
                    decrypted_block = decrypt_block(block)
                    p_decifrado.append(decrypted_block)
                write_decryption_results_to_file(decryption_file, p_decifrado)

            close_files(encryption_file, decryption_file)

            # Measure memory usage
            measure_memory_usage()

            # Store the results
            store_results(results, j)

# Save results to a CSV file
save_results_to_csv(results).
```

6 Experimental Results

To evaluate the performance and security of our algorithm, we conducted a series of experiments using various plaintexts and key sizes. The results demonstrate that our algorithm achieves robust encryption while maintaining computational efficiency.

6.1 Phase 1

The experimentation is carried out with ten national anthems, which are listed below:

1. Australia,
2. Canada,
3. Great Britain,
4. India,
5. Ireland,
6. Jamaica,
7. New Zealand,
8. Singapore,
9. South Africa,
10. USA.

These anthems were used for the first part of the experimentation. In this specific chapter, the experimentation consists of a single round of the HAJ algorithm. That is, encryption and decryption are performed only once. The particularity is that a matrix of the size of the plaintext is generated.

The length of each anthem is as shown in table 1.

For example, the national anthem of the USA consists of 1705 characters with all the current verses and choruses, without being processed through the algorithm to make it a multiple.

Therefore, a random matrix K of 1705×1705 elements ranging from 0 to 256 is generated, which is multiplied by the plaintext to generate the ciphertext. For this particular example described here, it would be the anthem of the USA. The tables 2, 3, and 4 show the average, standard deviation, maximum, and minimum values of the execution time expressed in seconds, as well as the memory usage represented in Mb, from the thirty runs performed for each anthem where the key is the same size as the plaintext.

6.2 Phase 2 Results

The anthems used for this phase are the same that we used on subsection 6.1. The difference is that the length of each anthem changed because we made them multiple of 300 for experimental purposes. The length of each anthem are as show table 5:

Table 1. Length of each anthem

Anthem	Length
Australia	620
Canadá	316
Gran Bretaña	496
India	336
Irlanda	800
Jamaica	591
Nueva Zelanda	1190
Singapur	204
Sudáfrica	226
USA	1705

The tables 6, 7, and 8 show the execution for each anthem dividing the plaintext into blocks of 3 and creating 21 keys the size of each block, this is because there is one key for each block and each round. The tables show as well the average, standard deviation, maximum, and minimum values of the execution time and memory usage.

The tables 9, 10, and 11 show the execution for each anthem dividing the plaintext into blocks of 30 and creating 210 keys the size of each block, this is because there is one key for each block and each round. The tables show as well the average, standard deviation, maximum, and minimum values of the execution time and memory usage.

The tables 12, 13, and 14 show the execution for each anthem dividing the plaintext into blocks of 300 and creating 2100 keys the size of each block, this is because there is one key for each block and each round. The tables show as well the average, standard deviation, maximum, and minimum values of the execution time and memory usage.

6.3 Comparison between the Performance of the Devices

In general, we can see that our algorithm is very stable in memory use. Each device, where the algorithm was executed, had very similar memory usage consumption.

The difference was in the execution time, with the MacBook Air being the device with the worst performance. The Windows device and the online version had very similar execution times.

Figure 1 show the average execution time and memory usage of the runs for a single round of the HAJ. Where the key is the same size as the plaintext. In

Table 2. Mac Runs Summary

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	13.3370	5.9330
	SD	2.2320	0.0070
	Max	21.8156	5.9398
	Min	11.7482	5.9011
Canada	Average	2.2672	1.6134
	SD	0.1914	0.0070
	Max	2.7470	1.6202
	Min	2.0245	1.5815
Great Britain	Average	7.9106	3.8298
	SD	1.6691	0.0070
	Max	15.6925	3.8366
	Min	6.6204	3.7979
India	Average	2.5626	1.7630
	SD	0.1623	0.0068
	Max	2.9756	1.7697
	Min	2.3115	1.7323
Ireland	Average	27.7280	9.8431
	SD	3.9408	0.0070
	Max	39.7620	9.8499
	Min	24.6128	9.8112
Jamaica	Average	11.6885	5.3762
	SD	1.3711	0.0068
	Max	15.1518	5.3830
	Min	10.1736	5.3455
New Zealand	Average	92.7189	21.7070
	SD	18.0516	0.0070
	Max	149.5322	21.7138
	Min	82.6005	21.6751
Singapore	Average	0.7071	0.6921
	SD	0.1679	0.0070
	Max	1.3809	0.6989
	Min	0.5553	0.6602
South Africa	Average	0.7658	0.8320
	SD	0.0520	0.0070
	Max	0.9386	0.8388
	Min	0.7127	0.8001
USA	Average	280.4113	44.5373
	SD	59.3591	0.0070
	Max	576.2381	44.5441
	Min	255.1275	44.5054

Table 3. Runs Summary

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	8.2658	5.9330
	SD	0.1805	0.0070
	Max	8.5023	5.9398
	Min	7.9580	5.9011
Canada	Average	1.0878	1.5841
	SD	0.0289	0.0070
	Max	1.1596	1.5909
	Min	1.0596	1.5522
Great Britain	Average	3.9785	3.8298
	SD	0.0473	0.0070
	Max	4.1067	3.8366
	Min	3.9137	3.7979
India	Average	1.2708	1.7630
	SD	0.0165	0.0068
	Max	1.3227	1.7697
	Min	1.2483	1.7323
Ireland	Average	17.6643	9.8431
	SD	0.3715	0.0070
	Max	18.3467	9.8499
	Min	17.0091	9.8112
Jamaica	Average	7.0676	5.3762
	SD	0.1952	0.0068
	Max	7.5738	5.3830
	Min	6.7943	5.3455
New Zealand	Average	61.2749	21.7070
	SD	2.0358	0.0070
	Max	64.1252	21.7138
	Min	57.9498	21.6751
Singapore	Average	0.3318	0.6921
	SD	0.0617	0.0070
	Max	0.6527	0.6989
	Min	0.3075	0.6602
South Africa	Average	0.4134	0.8321
	SD	0.0107	0.0070
	Max	0.4581	0.8388
	Min	0.4010	0.8001
USA	Average	198.1801	44.5373
	SD	11.1756	0.0070
	Max	213.1193	44.5441
	Min	176.1961	44.5054

Table 4. Online Runs Summary

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	5.4928	5.9330
	SD	0.0870	0.0070
	Max	5.7460	5.9398
	Min	5.4067	5.9011
Canada	Average	0.9267	1.6134
	SD	0.0317	0.0070
	Max	1.0079	1.6202
	Min	0.8720	1.5815
Great Britain	Average	3.0410	3.8298
	SD	0.0336	0.0070
	Max	3.1122	3.8366
	Min	2.9755	3.7979
India	Average	1.0303	1.7630
	SD	0.0250	0.0068
	Max	1.0771	1.7697
	Min	0.9749	1.7323
Ireland	Average	11.6492	9.8431
	SD	0.2980	0.0070
	Max	12.6112	9.8499
	Min	11.3622	9.8112
Jamaica	Average	5.0064	5.3762
	SD	0.0669	0.0068
	Max	5.2021	5.3830
	Min	4.8887	5.3455
New Zealand	Average	56.5294	29.3796
	SD	1.1452	0.0070
	Max	59.4291	29.3864
	Min	54.8559	29.3477
Singapore	Average	0.2881	0.6921
	SD	0.0098	0.0070
	Max	0.3312	0.6989
	Min	0.2803	0.6602
South Africa	Average	0.3697	0.8321
	SD	0.0113	0.0070
	Max	0.4188	0.8388
	Min	0.3588	0.8001
USA	Average	103.6944	44.5373
	SD	3.7643	0.0070
	Max	110.1514	44.5441
	Min	98.1536	44.5054

Table 5. Length of each anthem for blocks

Anthem	Length
Australia	900
Canadá	600
Gran Bretaña	600
India	600
Irlanda	900
Jamaica	600
Nueva Zelanda	1200
Singapur	300
Sudáfrica	300
USA	1800

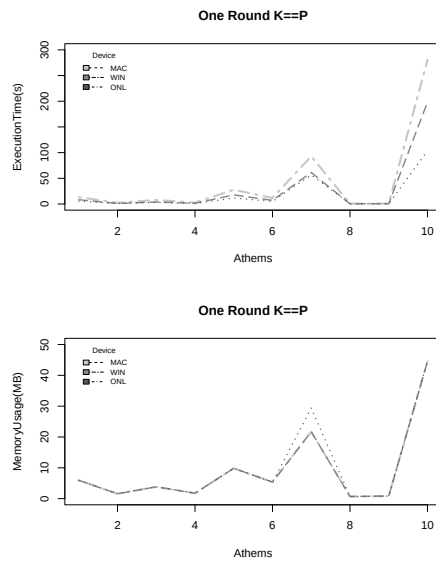


Fig. 1. Average accuracy across 30 runs.

Table 6. Runs per block (three) Mac.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	36.4588	15.8624
	SD	1.4329	0.0089
	Max	40.4950	15.8695
	Min	34.5879	15.8193
Canada	Average	11.5354	7.0801
	SD	0.5840	0.0089
	Max	13.0800	7.0873
	Min	10.9522	7.0371
Great Britain	Average	11.4738	7.0815
	SD	0.4413	0.0089
	Max	12.3882	7.0886
	Min	10.9528	7.0385
India	Average	11.5210	7.0803
	SD	0.5554	0.0089
	Max	13.4812	7.0874
	Min	10.9673	7.0372
Ireland	Average	36.5324	15.8637
	SD	1.2092	0.0089
	Max	39.1249	15.8709
	Min	34.5200	15.8207
Jamaica	Average	11.2850	7.0822
	SD	0.4914	0.0089
	Max	12.6359	7.0894
	Min	10.7407	7.0392
New Zealand	Average	83.7029	28.1561
	SD	3.1496	0.0089
	Max	93.8784	28.1633
	Min	79.3689	28.1131
Singapore	Average	1.5835	1.8089
	SD	0.1966	0.0089
	Max	2.2932	1.8160
	Min	1.4239	1.7658
South Africa	Average	1.6398	1.8091
	SD	0.0630	0.0089
	Max	1.8269	1.8162
	Min	1.4895	1.7660
USA	Average	277.0724	63.2675
	SD	17.8858	0.0089
	Max	301.9495	63.2746
	Min	246.4505	63.2244

the x-axis, we can see the numbers from 1 to 10. These numbers are the anthems as listed in table 6.1.

Figure 2 show the average execution time and memory usage of the runs by three blocks for the plaintext and running it seven times. Where the key is the

Table 7. Runs per block (three) Windows.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	22.2355	15.8624
	SD	0.0704	0.0089
	Max	22.3828	15.8695
	Min	22.0792	15.8193
Canada	Average	7.1483	7.0801
	SD	0.8943	0.0089
	Max	11.5577	7.0873
	Min	6.8720	7.0371
Great Britain	Average	30.0705	7.0815
	SD	124.5569	0.0089
	Max	689.5307	7.0886
	Min	6.9026	7.0385
India	Average	6.9861	7.0803
	SD	0.1767	0.0089
	Max	7.8852	7.0874
	Min	6.9125	7.0372
Ireland	Average	22.2087	15.8637
	SD	0.0743	0.0089
	Max	22.4107	15.8709
	Min	22.0785	15.8207
Jamaica	Average	6.9500	7.0822
	SD	0.0384	0.0089
	Max	7.0793	7.0894
	Min	6.9087	7.0392
New Zealand	Average	52.8028	28.1561
	SD	3.2683	0.0089
	Max	62.2822	28.1633
	Min	50.2036	28.1131
Singapore	Average	0.9744	1.8089
	SD	0.0135	0.0089
	Max	1.0212	1.8160
	Min	0.9550	1.7658
South Africa	Average	0.9827	1.8091
	SD	0.0106	0.0089
	Max	1.0160	1.8162
	Min	0.9656	1.7660
USA	Average	180.5225	63.2675
	SD	51.7192	0.0089
	Max	453.0388	63.2746
	Min	161.4610	63.2244

size of each block. In the x-axis, we can see the numbers from 1 to 10. These numbers are the anthems as listed in table 6.1.

Figure 3 show the average execution time and memory usage of the runs by thirty blocks for the plaintext and running it seven times. Where the key is the

Table 8. Runs per block (three) Online.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	16.1643	15.8624
	SD	1.0832	0.0089
	Max	19.7402	15.8695
	Min	15.4277	15.8193
Canada	Average	6.4902	7.0801
	SD	0.9334	0.0089
	Max	9.0094	7.0873
	Min	6.0004	7.0371
Great Britain	Average	6.4617	7.0815
	SD	0.7231	0.0089
	Max	8.1688	7.0886
	Min	5.9552	7.0385
India	Average	6.0322	7.0803
	SD	0.0324	0.0089
	Max	6.1227	7.0874
	Min	5.9866	7.0372
Ireland	Average	15.6482	15.8637
	SD	0.2025	0.0089
	Max	16.2677	15.8709
	Min	15.4033	15.8207
Jamaica	Average	6.6892	7.0822
	SD	0.9726	0.0089
	Max	8.6431	7.0894
	Min	5.9876	7.0392
New Zealand	Average	34.5667	28.1574
	SD	0.4773	0.0091
	Max	36.2345	28.1646
	Min	34.0755	28.1131
Singapore	Average	0.9882	1.8089
	SD	0.0109	0.0089
	Max	1.0342	1.8160
	Min	0.9801	1.7658
South Africa	Average	0.9895	1.8091
	SD	0.0083	0.0089
	Max	1.0206	1.8162
	Min	0.9815	1.7660
USA	Average	113.6309	63.2687
	SD	1.6772	0.0091
	Max	116.8265	63.2759
	Min	111.2799	63.2244

size of each block. In the x-axis, we can see the numbers from 1 to 10. These numbers are the anthems as listed in table 6.1.

Figure 4 show the average execution time and memory usage of the runs by three hundred blocks for the plaintext and running it seven times. Where the

Table 9. Runs per block (thirty) Mac.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	0.7155	1.5414
	SD	0.8525	0.0089
	Max	5.1868	1.5485
	Min	0.4533	1.4983
Canada	Average	0.4028	0.7251
	SD	0.9142	0.0089
	Max	5.2114	0.7322
	Min	0.1600	0.6820
Great Britain	Average	0.3778	0.7265
	SD	1.0161	0.0089
	Max	5.7564	0.7336
	Min	0.1704	0.6834
India	Average	0.3017	0.7252
	SD	0.4158	0.0089
	Max	2.4866	0.7324
	Min	0.1741	0.6822
Ireland	Average	0.7427	1.5427
	SD	0.9062	0.0089
	Max	5.4912	1.5499
	Min	0.4681	1.4997
Jamaica	Average	0.2054	0.7272
	SD	0.0472	0.0089
	Max	0.3668	0.7343
	Min	0.1560	0.6841
New Zealand	Average	1.2550	2.6832
	SD	0.3763	0.0089
	Max	3.2087	2.6903
	Min	1.1071	2.6401
Singapore	Average	0.2285	0.2338
	SD	0.7445	0.0089
	Max	4.1370	0.2409
	Min	0.0377	0.1907
South Africa	Average	0.0624	0.2339
	SD	0.0138	0.0089
	Max	0.0919	0.2411
	Min	0.0424	0.1909
USA	Average	5.1072	5.9324
	SD	1.6908	0.0089
	Max	9.7633	5.9396
	Min	3.1777	5.8894

Table 10. Runs per block (thirty) Windows.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	0.3077	1.5414
	SD	0.0204	0.0089
	Max	0.3641	1.5485
	Min	0.2825	1.4983
Canada	Average	0.1028	0.7251
	SD	0.0098	0.0089
	Max	0.1400	0.7322
	Min	0.0934	0.682
Great Britain	Average	0.1063	0.7265
	SD	0.0165	0.0089
	Max	0.1791	0.7336
	Min	0.0938	0.6834
India	Average	0.1048	0.7252
	SD	0.0157	0.0089
	Max	0.1687	0.7324
	Min	0.0929	0.6822
Ireland	Average	0.298	1.5427
	SD	0.0135	0.0089
	Max	0.3429	1.5499
	Min	0.2849	1.4997
Jamaica	Average	0.1042	0.7272
	SD	0.0115	0.0089
	Max	0.1449	0.7343
	Min	0.0926	0.6841
New Zealand	Average	0.6614	2.6832
	SD	0.0136	0.0089
	Max	0.716	2.6903
	Min	0.6445	2.6401
Singapore	Average	0.0287	0.2338
	SD	0.008	0.0089
	Max	0.0622	0.2409
	Min	0.02	0.1907
South Africa	Average	0.0277	0.2339
	SD	0.007	0.0089
	Max	0.052	0.2411
	Min	0.0184	0.1909
USA	Average	2.1344	5.9324
	SD	0.0177	0.0089
	Max	2.1894	5.9396
	Min	2.112	5.8894

key is the size of each block. In the x-axis, we can see the numbers from 1 to 10. These numbers are the anthems as listed in table 6.1.

Table 11. Runs per block (thirty) Online.

Anthem	Parameters	ExecutionTime(s)	MemoryUsage(MB)
Australia	Average	0.3089	1.5414
	SD	0.0091	0.0089
	Max	0.3470	1.5485
	Min	0.3034	1.4983
Canada	Average	0.1070	0.7251
	SD	0.0069	0.0089
	Max	0.1421	0.7322
	Min	0.1035	0.6820
Great Britain	Average	0.1089	0.7265
	SD	0.0093	0.0089
	Max	0.1499	0.7336
	Min	0.1043	0.6834
India	Average	0.1091	0.7252
	SD	0.0124	0.0089
	Max	0.1695	0.7324
	Min	0.1036	0.6822
Ireland	Average	0.3073	1.5427
	SD	0.0084	0.0089
	Max	0.3432	1.5499
	Min	0.3026	1.4997
Jamaica	Average	0.1083	0.7272
	SD	0.0080	0.0089
	Max	0.1460	0.7343
	Min	0.1042	0.6841
New Zealand	Average	0.6830	2.6832
	SD	0.0124	0.0089
	Max	0.7256	2.6903
	Min	0.6724	2.6401
Singapore	Average	0.0289	0.2338
	SD	0.0067	0.0089
	Max	0.0602	0.2409
	Min	0.0254	0.1907
South Africa	Average	0.0283	0.2339
	SD	0.0084	0.0089
	Max	0.0590	0.2411
	Min	0.0246	0.1909
USA	Average	2.1718	5.9324
	SD	0.0105	0.0089
	Max	2.2157	5.9396
	Min	2.1613	5.8894

7 Conclusions

The HAJ algorithm constituted an initial contribution to symmetric cryptography based on matrix operations by combining the classical Hill cipher with

Table 12. Runs per block (three hundred) for Mac.

Anthem	Parameters	Execution Time (s)	Memory Usage (MB)
Australia	Average	0.0819	0.4172
	SD	0.0291	0.0089
	Max	0.1932	0.4243
	Min	0.0573	0.3742
Canada	Average	0.0771	0.3295
	SD	0.0241	0.0089
	Max	0.1360	0.3367
	Min	0.0533	0.2865
Great Britain	Average	0.0762	0.3309
	SD	0.0347	0.0089
	Max	0.1894	0.3381
	Min	0.0441	0.2879
India	Average	0.0645	0.3297
	SD	0.0230	0.0089
	Max	0.1288	0.3368
	Min	0.0470	0.2866
Ireland	Average	0.0780	0.4186
	SD	0.0301	0.0089
	Max	0.1845	0.4257
	Min	0.0524	0.3755
Jamaica	Average	0.0733	0.3316
	SD	0.0232	0.0089
	Max	0.1178	0.3388
	Min	0.0444	0.2886
New Zealand	Average	0.1009	0.5390
	SD	0.0396	0.0088
	Max	0.2070	0.5461
	Min	0.0766	0.4959
Singapore	Average	0.0654	0.2754
	SD	0.0210	0.0089
	Max	0.1293	0.2826
	Min	0.0424	0.2324
South Africa	Average	0.0604	0.2756
	SD	0.0253	0.0089
	Max	0.1490	0.2827
	Min	0.0389	0.2325
USA	Average	0.1395	0.8740
	SD	0.0418	0.0088
	Max	0.2806	0.8811
	Min	0.1132	0.8309

Table 13. Runs per block (three hundred) for Windows.

Anthem	Parameters	ExecutionTime(s)	MemoryUsage(MB)
Australia	Average	0.0384	0.4172
	SD	0.0095	0.0089
	Max	0.0588	0.4243
	Min	0.0271	0.3742
Canada	Average	0.0356	0.3295
	SD	0.0083	0.0089
	Max	0.0670	0.3367
	Min	0.0254	0.2865
Gran Bretaña	Average	0.0348	0.3309
	SD	0.0104	0.0089
	Max	0.0673	0.3381
	Min	0.0232	0.2879
India	Average	0.0338	0.3297
	SD	0.0153	0.0089
	Max	0.1033	0.3368
	Min	0.0235	0.2866
Ireland	Average	0.0374	0.4186
	SD	0.0108	0.0089
	Max	0.0711	0.4257
	Min	0.0280	0.3755
Jamaica	Average	0.0322	0.3316
	SD	0.0061	0.0089
	Max	0.0462	0.3388
	Min	0.0238	0.2886
New Zealand	Average	0.0457	0.5390
	SD	0.0111	0.0088
	Max	0.0790	0.5461
	Min	0.0335	0.4959
Singapore	Average	0.0285	0.2754
	SD	0.0065	0.0089
	Max	0.0550	0.2826
	Min	0.0210	0.2324
South Africa	Average	0.0251	0.2756
	SD	0.0059	0.0089
	Max	0.0414	0.2827
	Min	0.0170	0.2325
USA	Average	0.0785	0.8740
	SD	0.0285	0.0088
	Max	0.2104	0.8811
	Min	0.0610	0.8309

the Gauss-Jacques method for modular matrix inversion. The principal novelty of this initial work was the integration of these two mathematical procedures into a single encryption/decryption framework, together with a multi-round and block-based implementation.

Table 14. Runs per block (three hundred) Online.

Anthem	Parameters	ExecutionTime(s)	MemoryUsage(MB)
Australia	Average	0.0355	0.4172
	SD	0.0125	0.0089
	Max	0.1008	0.4243
	Min	0.0312	0.3742
Canada	Average	0.0308	0.3295
	SD	0.0147	0.0089
	Max	0.1064	0.3367
	Min	0.0261	0.2865
Great Britain	Average	0.0297	0.3309
	SD	0.0092	0.0089
	Max	0.0661	0.3381
	Min	0.0252	0.2879
India	Average	0.0283	0.3297
	SD	0.0079	0.0089
	Max	0.0697	0.3368
	Min	0.0258	0.2866
Ireland	Average	0.0337	0.4186
	SD	0.0058	0.0089
	Max	0.0615	0.4257
	Min	0.0303	0.3755
Jamaica	Average	0.0293	0.3316
	SD	0.0079	0.0089
	Max	0.0677	0.3388
	Min	0.0258	0.2886
New Zealand	Average	0.0426	0.5390
	SD	0.0106	0.0088
	Max	0.0782	0.5461
	Min	0.0378	0.4959
Singapore	Average	0.0243	0.2754
	SD	0.0034	0.0089
	Max	0.0416	0.2826
	Min	0.0223	0.2324
South Africa	Average	0.0246	0.2756
	SD	0.0063	0.0089
	Max	0.0579	0.2827
	Min	0.0224	0.2325
USA	Average	0.0705	0.8740
	SD	0.0095	0.0088
	Max	0.1158	0.8811
	Min	0.0637	0.8309

The results obtained from thirty-run experiments across different plaintext sizes and computational environments provided an initial characterization of the algorithm's computational behavior.

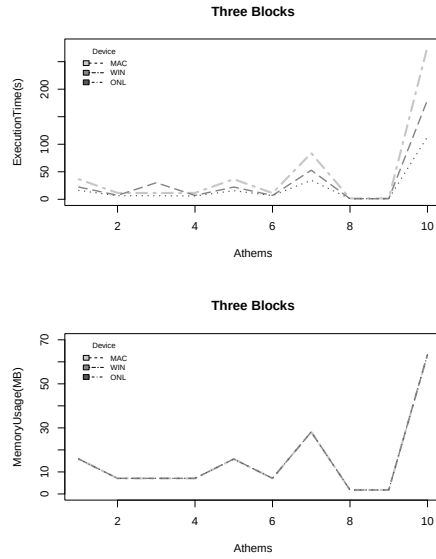


Fig. 2. Average accuracy across 30 runs.

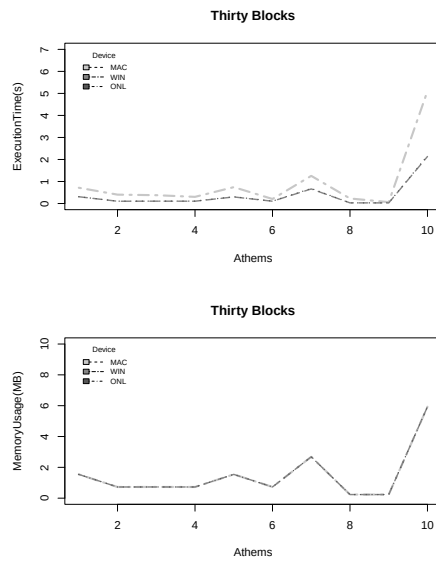


Fig. 3. Average accuracy across 30 runs.

The experimental results showed that memory consumption remained comparatively stable for a given configuration, whereas execution time was strongly

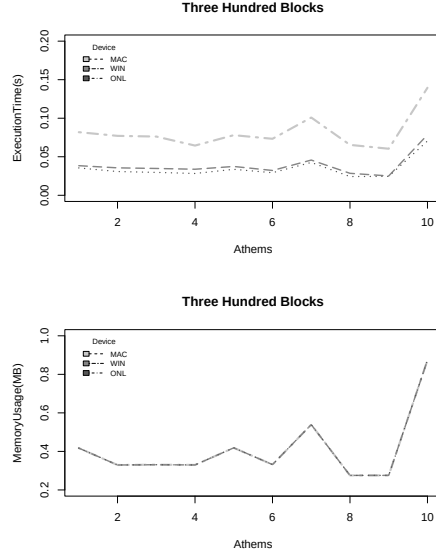


Fig. 4. Average accuracy across 30 runs.

influenced by plaintext length, matrix dimension, number of blocks, and computational platform.

An important observation was that the implementation was also able to operate with a matrix dimension of 1705×1705 in the experiments, demonstrating that the proposed construction could handle substantially large matrices within the evaluated computational environments.

This result provided a useful indication of the computational scalability of the approach, although matrix size alone could not be interpreted as evidence of cryptographic security, entropy, or randomness.

The experiments also showed the effect of changing the block configuration on the computational behavior of the algorithm. These observations provided practical evidence regarding the performance characteristics of HAJ and established an initial computational baseline for its further development. The results were interpreted as performance measurements rather than as a formal proof of cryptographic security.

The combination of the Gauss-Jacques method and the extended Euclidean algorithm provided a systematic procedure for computing the modular inverses required during decryption. This was particularly relevant because modular inversion represented an essential operation within the proposed matrix-based construction.

At the same time, the study primarily addressed algorithmic design and computational evaluation. A comprehensive security assessment remained necessary, including statistical randomness testing, differential and linear cryptanalysis,

known-plaintext and chosen-plaintext attacks, side-channel considerations, and a systematic analysis of key generation and management.

An important outcome of this research was that the matrix-based framework introduced by HAJ opened a broader line of investigation into the algebraic structure of cryptographic matrices. Thus, the contribution of HAJ was both concrete and foundational: it proposed a specific symmetric construction, evaluated its computational behavior across different configurations, demonstrated its operation with substantially large matrix dimensions, and motivated deeper investigation into the algebraic properties of matrix-based cryptographic primitives.

Future work should focus on formal cryptanalysis, optimization of matrix generation and inversion, larger and systematically selected parameter sets, standardized statistical testing, resistance to established attack models, implementation-level side-channel analysis, and the study of structured non-diagonalizable matrix constructions under classical and quantum attack models.

These investigations were necessary to determine whether the algebraic structures explored in this research could ultimately lead to cryptographic constructions with formally established security properties.

References

1. Stallings, W.: *Cryptography and Network Security: Principles and Practice*. Pearson Education (2017)
2. Paar, C., Pelzl, J.: *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Science & Business Media (2010)
3. Radhakrishnan, I., et al.: Efficiency and Security Evaluation of Lightweight Cryptographic Algorithms for Resource-Constrained IoT Devices. *Sensors*, 24(12), pp. 4008 (2024)
4. Sarker, K.U., et al.: A Systematic Review on Lightweight Security Algorithms for Resource-Constrained Systems. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* (2025). Doi: 10.1007/s43926-025-00150-4.
5. Wen, H., Lin, Y., Yang, L., Chen, R.: Cryptanalysis of an Image Encryption Scheme Using Variant Hill Cipher and Chaos. *Expert Systems with Applications*, 250, 123748 (2024). Doi:10.1016/j.eswa.2024.123748.
6. Teseleanu, G.: Security Analysis of an Image Encryption Scheme Based on a New Secure Variant of Hill Cipher and 1D Chaotic Maps. In: *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, SCITEPRESS, pp. 745–749 (2024). Doi:10.5220/0012356600003648.
7. Bufalo, M.: Some Properties of the Computation of the Modular Inverse with Applications in Cryptography. *Mathematics*, 11(4), pp. 70, (2023). Doi: 10.3390/math11040070.
8. Xu, G., Tian, Y., Yang, B.: On the Inversion Modulo a Power of an Integer. *arXiv Preprint*, arXiv:2506.02491, <https://arxiv.org/abs/2506.02491> (2025)
9. Rrghout, H., Kattass, M., Qobbi, Y., Benazzi, N., Jarjar, A.: Robust Image Encryption Algorithm Using a New Variant of Hill Cipher. In: *Proceedings of the 6th International Conference on NISS, ACM* (2023). Doi: 10.1145/3607720.3607750.

10. Alarcon, A.: Towards a Symmetric Crypto Algorithm: The HAJ. ITNG 2021 18th International Conference on Information Technology-New Generations (2021)
11. Alshammari, B.M., Guesmi, R., Guesmi, T., Alsaif, H., Alzamil, A.: Implementing a symmetric lightweight cryptosystem in highly constrained IoT devices by using a chaotic S-box. *Symmetry*, 131, p. 129, 2021.
12. Hill, L.S.: Cryptography in an Algebraic Alphabet. *The American Mathematical Monthly*, 36, pp. 306–312 (1929)
13. Schneier, B.: *Applied Cryptography: Protocols, Algorithms, and Source Code* in C. John Wiley & Sons (1996)
14. Ali, N.H.M., Hoobi, M.M., Saffo, D.F.: Development of Robust and Efficient Symmetric Random Keys Model based on the Latin Square Matrix. *Mesopotamian Journal of Cybersecurity*, 4(3), pp. 203–215 (2024)
15. Singh, S., Srivastava, S.: Cryptanalysis of Hill cipher using known plaintext attack. *International Journal of Scientific and Research Publications*, 4, pp. 1–4 (2014)
16. Cui, X., Zhang, H., Fang, X., Wang, Y., Wang, D., Fan, F., Shu, L.: A secret key classification framework of symmetric encryption algorithm based on deep transfer learning. *Applied Sciences*, 13(21), pp. 12025 (2023)
17. Agrawal, M., Gupta, D.: Cryptanalysis of Hill Cipher with Known Ciphertext Attack. *International Journal of Computer Applications*, 72, pp. 39–44 (2013)
18. Singh, S., Srivastava, S.: Cryptanalysis of Hill Cipher using Frequency Attack. *International Journal of Computer Applications*, 86, pp. 1–5 (2014)
19. Cantón, D.: Gauss-Jacques Method. <https://github.com/dCantonE/gauss-jacques> (2023)