Enhanced Pyjamask-128 with Key-Dependent Dynamic MixRows and HKDF-SHA-256-Based Round Key Generation for Secure IoT Data Encryption

Rasha Thamer Shawi

Department of Computer Science, College of Education, Mustansiriyah University, Baghdad, Iraq

[1]*Received: 12/09/2026; Accepted: 03/10/2026; Published: 28/10/2026*

Abstract

The Internet of Things (IoT) is in many current applications because connected devices send data to each other in different places - it is hard to keep IoT communication secure when devices don't have much memory or processing capacity. Because of this developers have to use lightweight cryptographic algorithms to protect data and keep resource use low. In this study we propose an altered Pyjamask-128 lightweight block cipher to encrypt IoT data securely. We changed the algorithm in two ways. We used Key Dependent Dynamic MixRows (KDDM), which has key dependent circular rotations in the diffusion layer. We used HKDF-SHA-256 to generate 15 round keys and 56 dynamic rotation parameters. The block size is still 128 bits and the algorithm still has 14 rounds. In our tests the altered algorithm had an average avalanche effect of 50.02% and a key sensitivity BER of 50.0195% - it passed all applied NIST SP 800-22 tests at the 0.01 significance level plus had more rapid early round diffusion. Encrypting took about 5.39% more time, decrypting took about 5.61% more time and the algorithm had 3.75% more memory consumption - those outcomes show the algorithm is more secure; people can test it more in IoT settings that have low resources.

Keywords*: The Internet of Things; lightweight cryptographic algorithms; Pyjamask-128; HKDF-SHA-256*

1. Introduction

The Internet of Things (IoT) is a system where devices connect to each other. Those devices collect and process but also send information in areas like healthcare, factory automation and sensor networks and transportation systems. But because people use more of those devices now, keeping data private and correct also safe from unauthorized users is hard. Many IoT devices have little processing power plus small memories but also limited battery life. Because of this using heavy cryptographic algorithms on them is difficult [1]. Researchers study lightweight cryptography to protect those small systems. The National Institute of Standards & Technology (NIST) has stated that algorithms for those devices must have both security and low resource use [2][3].

Lightweight block ciphers keep data private whilst using little computation and memory and hardware space. Pyjamask is a group of lightweight block ciphers based on the Substitution - Permutation Network (SPN) design [4]. Pyjamask has two versions - Pyjamask-96 and Pyjamask-128. Both have 128-bit secret keys and 14 rounds. Pyjamask-128 encrypts 128-bit blocks of data using AddRoundKey, SubBytes and MixRows steps in order. Those steps confuse and spread the data patterns, which block ciphers need to do to be safe [5]. Because Pyjamask-128 has those steps,

[1] *How to cite the article:* Sahwi R.T.; October 2026; Enhanced Pyjamask-128 with Key-Dependent Dynamic MixRows and HKDF-SHA-256-Based Round Key Generation for Secure IoT Data Encryption; *International Journal of Innovations in Scientific Engineering*, Jul-Dec 2026, Vol 24, 65-88

individuals test it for use in IoT devices. Though Pyjamask-128 has a set design, it is useful to study it more, how it spreads data patterns, generates round keys. The original MixRows step has fixed binary matrices; the way it spreads data doesn't change when the key changes. The original key schedule makes round keys using set steps that repeat. Because of those facts, we can study what happens when data spreading changes with the key and when we use other ways to make round keys - this includes how those changes affect differential and linear attributes and how round keys relate to each other. People need to measure the security plus the computing time when combining dynamic diffusion with keys that are made separately [6][7].To address this problem, we propose change Pyjamask-128 in two ways to encrypt IoT data. We add a Key Dependent Dynamic MixRows step - putting bit shifts for each round into the diffusion layer. The secret key sets those shift numbers; the diffusion step is different in each round. We use HKDF-SHA-256 to make round keys instead of the original key schedule. This method uses the extract-and-expand steps from RFC 5869. It makes 15 round keys of 128 bits and 56 shift numbers using separate steps.

Our algorithm has the original 128-bit block size and 14 rounds but it changes the diffusion step using the key and has a different way to make round keys. We test how well it works - measuring the avalanche effect, key sensitivity, randomness and entropy also speed. We also compare it to the original Pyjamask-128 to see its security attributes and its performance costs but also if it works well on small IoT devices.

The major contributions of this work are summarized as follows:

- We modified the Pyjamask-128 cipher - It has Key Dependent Dynamic MixRows and round keys that come from HKDF-SHA-256 so that people can use it to secure IoT devices.
- A dynamic diffusion mechanism using key-dependent rotations and an HKDF-SHA-256-based approach to generate 15 round keys and 56 rotation parameters.
- the evaluated the design and compared it with the original Pyjamask-128. In those tests the avalanche effect was larger, the key sensitivity was higher, the bits diffused more in early rounds and the information entropy had a higher value.

The rest of the paper has this structure - Section Two is a review of prior literature, Section Three is about the theory of the original Pyjamask-128 algorithm, Section Four is about the modified algorithm, Section Five is about the test results and Section Six is the conclusion.

2. Literature Review

Deng et al. [8] studied how linear diffusion layers work mathematically in lightweight block ciphers, which had Pyjamask-128 amongst them. The authors had a goal to find the lowest cycle lengths of linear transformations that people use often - they also looked at how Circular Boolean Matrix (CBM) operations relate to Rotational-XOR operations mathematically. The paper showed those operations are equal and the authors had the linear layer of Pyjamask-128 as an example to explain this - this work showed the algebra and function of diffusion parts in ciphers. The paper helps individuals understand how linear transformations work mathematically but it is theoretical and doesn't make the internal parts of Pyjamask-128 work better. The study has no MixRows transformations that change based on keys - it has no new way to make round keys. Because of this it is possible for researchers to study diffusion structures that change plus different ways to schedule keys in the future.

Park et al. [9] described a way to implement the block cipher Pyjamask that uses less computing resources than previous versions. Pyjamask was an entry for the lightweight cryptography standardization process from the National Institute of Standards & Technology (NIST). This method changes the linear layer and the key schedule so that the cipher uses fewer XOR operations when it encrypts data. In the study the linear layer had 7.81% fewer XOR operations and the key schedule had 9.44% fewer XOR operations. Because of those changes, the whole implementation had 5.16% fewer

operations than before. While this design requires less work from a computer, it only makes the code simpler to build - it doesn't add a diffusion mechanism that depends on the key and it doesn't derive cryptographic keys in a new way. Because of this researchers can still test different designs for the unchanging linear transformation plus the original key scheduling structure.

Cui et al. [10] tested how secure Pyjamask-96 and Pyjamask-128 with fewer rounds are when using integral cryptanalysis - they made integral distinguishers and recovered keys to see how well the cipher stops attacks - they had an 11-round integral attack on Pyjamask-128, which they made - adding rounds to a 9-round integral distinguisher. The results help people see how secure Pyjamask is when it has fewer rounds. But the paper is only about testing attacks and the authors didn't suggest changes to the design to make diffusion or round key generation better. The attacks are only for versions with fewer rounds - they aren't a way to break all 14 rounds of Pyjamask-128.

Tian and Hu [11] had a new method to search for integral distinguishers that work best - using how division property propagates in S-boxes - this method made the calculation needed to search for integral characteristics take less time. They used this technique on the Pyjamask and uBlock block ciphers. The researchers had a 9-round integral distinguisher for Pyjamask-128 and a 10-round integral distinguisher for Pyjamask-96. Those results show how Pyjamask diffuses data and resists integral cryptanalysis. The change is in the search algorithm used for cryptanalysis plus not in how the encryption is built. The study had no dynamic linear transformations and it didn't change how the cipher creates round keys to make the cipher's design better.

Liu Ya et al. [12] studied how secure the block cipher Pyjamask is - using impossible differential cryptanalysis, which is designed for devices that have few resources. The researchers found characteristics of impossible differentials so that they could see how the cipher resists attacks that are differential. This work helps people know the security attributes of Pyjamask when individuals use it in cryptography and it helps people know the points that are weak when the cipher has fewer rounds. The analysis gives facts about how secure the algorithm is but the text is about attacks from cryptanalysis instead of how to make the parts that encrypt inside the cipher better. In addition the authors had no changes for the MixRows transformation and they had no system that was better to generate keys for each round. Because of this it is possible for individuals in future research to make and test other structures for diffusion plus other ways to schedule keys.

Goudarzi et al. [13] made a block cipher that's Pyjamask - It has a small footprint, it authenticates data whilst encrypting it and it resists side channel attacks because masking it takes little time and memory. The design has Pyjamask-96 and Pyjamask-128. Both have a substitution - permutation network (SPN) structure that substitutes nonlinearly and diffuses linearly. The authors had an emphasis on masks that operate fast plus they showed the algorithm is usable on computers with limited resources. In their tests Pyjamask-128 was fast and protected on embedded processors, e.g. ARM Cortex-M4 platforms. The cipher has a fast, small structure but the base version has linear transformations that are fixed in advance and a key schedule that doesn't change. In addition the authors had no tests for dynamic MixRows transformations that change based on keys, nor did they have tests for making round keys with HKDF-SHA-256.

3.Theoretical background

Original Pyjamask-128 Block Cipher

Pyjamask-128 is a lightweight symmetric block cipher designed by Goudarzi et al. (2019) and subsequently described in detail in their published work in 2020, designed to provide efficient software implementation while facilitating protection against side-channel attacks through high-order masking techniques. The cipher employs a substitution–mixing architecture, processes 128-bit plaintext blocks using a 128-bit secret key, and performs 14 encryption rounds. Each round consists of three fundamental transformations: AddRoundKey, SubBytes, and MixRows. After completing the 14 rounds, an additional AddRoundKey operation is performed to generate the final ciphertext[14],[15].

The internal state is organized into four 32-bit rows, forming a 128-bit binary state represented by Equation (1):

$$S = \begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{bmatrix}, \qquad R_i \in \{0,1\}^{32} \ldots\ldots (1)$$

where R_i denotes the i-th 32-bit row of the internal state. The encryption transformations are described in the following subsections.

2.1. AddRoundKey Transformation

The AddRoundKey transformation combines the internal state with a 128-bit round key using the bitwise exclusive OR (XOR) operation. This step introduces the secret-key-dependent component into each encryption round and is mathematically expressed in Equation (2):

$$S_r^{(1)} = S_r^{(0)} \oplus K_r \ldots\ldots (2)$$

where $S_r^{(0)}$ represents the input state of round r, K_r denotes the corresponding 128-bit round key, and $\oplus$ represents bitwise XOR.

2.2. SubBytes Transformation

The SubBytes transformation introduces nonlinearity into the encryption process by applying a 4-bit substitution box, denoted as S_4, independently to each of the 32 columns of the internal state. The transformation is expressed as:

$$S_r^{(2)} = \text{SubBytes}\left(S_r^{(1)}\right) \ldots\ldots (3)$$

The original Pyjamask-128 substitution box is defined as follows:

$$S_4 = \begin{bmatrix} 2 & D & 3 & 9 & 7 & B & A & 6 \\ E & 0 & F & 4 & 8 & 5 & 1 & C \end{bmatrix} \ldots\ldots\ldots (4)$$

where the hexadecimal entries correspond to the substitution outputs for the input values from 0 to 15. This nonlinear transformation enhances confusion while maintaining a low number of nonlinear operations, supporting efficient masked implementations.

2.3. MixRows Transformation

The MixRows transformation provides linear diffusion across the internal state. Each 32-bit row is independently multiplied by a predefined 32×32 binary circulant matrix over the finite field $GF(2)$. This transformation is expressed in Equation (5):

$$R_i' = M_i R_i, \qquad i \in \{0,1,2,3\} \ldots\ldots (5)$$

where R_i represents the input row, R_i' denotes the transformed row, and M_i is the corresponding fixed binary circulant matrix. All matrix operations are performed over $GF(2)$.

The resulting state is expressed as:

$$S_r^{(3)} = \begin{bmatrix} M_0 R_0 \\ M_1 R_1 \\ M_2 R_2 \\ M_3 R_3 \end{bmatrix} \ldots\ldots (6)$$

The original Pyjamask-128 employs four distinct and predefined diffusion matrices, M_0, M_1, M_2, and M_3, which remain constant throughout the encryption process. These matrices are designed to achieve strong diffusion while maintaining

efficient implementation through circulant matrix operations. Because the original diffusion matrices are fixed and independent of the encryption key, the MixRows transformation provides a suitable foundation for investigating key-dependent dynamic diffusion mechanisms[16],[17].

2.4. Original Key Schedule

The original Pyjamask-128 key schedule derives 15 round keys from a 128-bit master key. Each round key consists of 128 bits, with 14 keys used during the encryption rounds and one additional key used for the final post-whitening operation. The key schedule consists of three sequential transformations: MixColumns, MixAndRotateRows, and AddConstant.

MixColumns: Each 4-bit column of the key state is multiplied by a predefined binary matrix over $GF(2)$, as expressed in Equation (7):

$$C_j' = M_C C_j \ldots..(7)$$

where the transformation matrix M_C is defined as:

$$M_C = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \ldots\ldots.(9)$$

This transformation distributes key-bit dependencies among the four rows of the key state.

MixAndRotateRows: The first 32-bit row is transformed using a predefined 32×32 binary circulant matrix M_K, while the remaining three rows undergo fixed circular left rotations. These operations are mathematically represented as:

$$\begin{aligned} R_0' &= M_K R_0, \\ R_1' &= \mathrm{ROTL}_{32}(R_1, 8), \\ R_2' &= \mathrm{ROTL}_{32}(R_2, 15), \\ R_3' &= \mathrm{ROTL}_{32}(R_3, 18) \end{aligned}$$

where ROTL_{32} denotes a circular left rotation of a 32-bit word by the specified number of positions. The matrix M_K used in the key schedule is distinct from the four diffusion matrices employed in the MixRows transformation.

AddConstant: A round-dependent constant is introduced into the key state through XOR operations. The constant consists of a fixed 28-bit hexadecimal value, 0x243F6A8, combined with a 4-bit round counter. This step distinguishes successive round keys and helps reduce structural symmetries within the key expansion process. The four bytes of the round constant are XORed into the key state: the three most significant bytes are applied to the most significant bytes of rows R3, R2, and R1, respectively, while the least significant byte, containing the round counter, is applied to the least significant byte of R0.

The complete key schedule is summarized by Equation (10):

$$K_{r+1} = \mathrm{AddConstant}_{r+1}\Big(\mathrm{MixAndRotateRows}\big(\mathrm{MixColumns}(K_r)\big)\Big) \ldots..(10)$$

where K_0 represents the original 128-bit master key, and K_{r+1} denotes the generated key state for the subsequent round. The original key schedule relies entirely on linear transformations and round-dependent constant addition, rather than cryptographic hash-based key derivation. Although this design supports efficient implementation, alternative key generation mechanisms may be investigated to introduce different key-dependent transformation properties. The structure of the original Pyjamask-128 key schedule, including MixColumns, MixAndRotateRows, and AddConstant, is illustrated in Fig. 1.

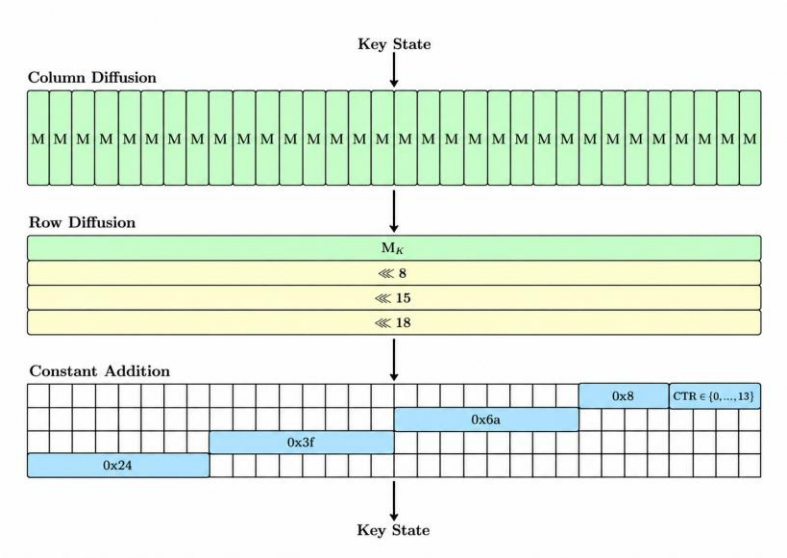


Fig. 1. Original Pyjamask-128 Key Schedule Structure

2.5. Overall Encryption Process

The complete encryption process consists of 14 rounds, each applying AddRoundKey, SubBytes, and MixRows in sequence. The overall round transformation is represented by Equation (11):

$$S_{r+1} = \text{MixRows}\big(\text{SubBytes}(S_r \oplus K_r)\big) \ldots\ldots(11)$$

where $r = 0,1, \ldots, 13$ and $S_0 = P$, with P denoting the 128-bit plaintext block.

After completing the 14 rounds, a final AddRoundKey transformation is applied to generate the ciphertext:

$$C = S_{14} \oplus K_{14} \ldots\ldots(12)$$

where C denotes the resulting 128-bit ciphertext. The three main transformations of the original Pyjamask-128 encryption round, namely AddRoundKey, SubBytes, and MixRows, are illustrated in Fig. 2.

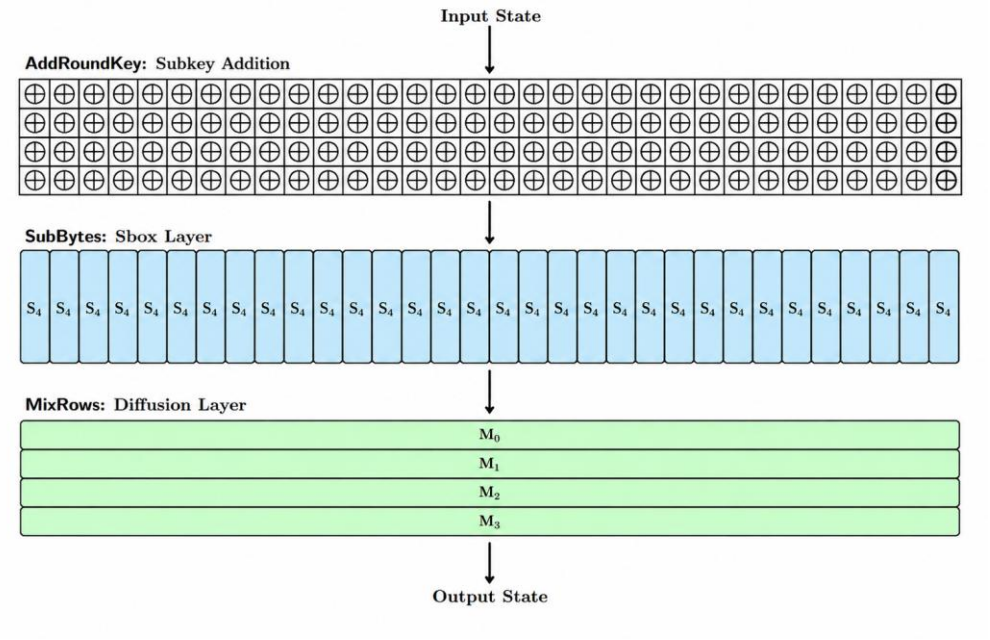


Fig. 2. Round Function of the Original Pyjamask-128

2.6. Decryption Process

The decryption process reverses the encryption transformations using the inverse operations in reverse round order. Initially, the final post-whitening key is removed from the ciphertext:

$S_{14} = C \oplus K_{14}$…..(13)

Subsequently, the inverse MixRows and inverse SubBytes transformations are applied, followed by the corresponding round-key addition. The inverse round operation is expressed as:

$S_r = \text{InvSubBytes}\big(\text{InvMixRows}(S_{r+1})\big) \oplus K_r$……(14)

where $r = 13,12, \dots ,0$.

After completing all inverse rounds, the original plaintext is recovered:

$P = S_0$…..(15)

The inverse MixRows operation uses the inverse matrices M_i^{-1}, while the inverse SubBytes transformation employs the inverse substitution box S_4^{-1}, ensuring correct recovery of the original plaintext[18].

4. Proposed Algorithm

We present a modified version of the Pyjamask-128 block cipher with small resource requirements to make its security stronger - changing the diffusion layer and the way it generates round keys. The cipher still has a block size of 128 bits, a secret key of 128 bits and 14 rounds. Fig. 1 shows this design has two parts. In the first part, the original diffusion layer has a Key Dependent Dynamic MixRows transformation that shifts bits in a circle based on the key. In the second part, the cipher uses an HKDF-SHA-256-Based Round Key Generation process to produce 15 round keys and 56 dynamic rotation parameters. The following subsections describe those two parts.

4.1 Key-Dependent Dynamic MixRows

The diffusion layer is a basic part of a block cipher because it spreads changes from the input bits to the internal state. Designers usually put fixed linear transformations in lightweight ciphers because they know the cost to build them and they work quickly. A fixed diffusion mapping is the same for every secret key and round number. Because of this we examine a diffusion transformation that changes with the key. We want to add changes based on the key but we also want to keep the 128-bit state and the 14 rounds. This study presents Key Dependent Dynamic MixRows (KDDM). This method has the original invertible MixRows matrices plus it rotates the bits in a circle for each 32-bit row after the linear transformation. The system gets the rotation amounts from the 128-bit master key - using the HKDF-SHA-256 process from Section 4.2. Different identifiers keep the round keys apart from the rotation parameters. The values are different for each round index The resulting transformation remains invertible, permitting exact decryption.

In round r the input state has 128 bits and is 4 rows of 32 bits

$$S^{(r)} = (S_0^{(r)}, S_1^{(r)}, S_2^{(r)}, S_3^{(r)}), \quad S_i^{(r)} \in GF(2)^{32} \quad (16)$$

For each row i ∈ {0, 1, 2, 3}, the corresponding original binary diffusion matrix M_i is applied over GF(2):

$$Y_i^{(r)} = M_i S_i^{(r)}, \quad M_i \in GF(2)^{32x32} \quad (17)$$

One gets a rotation amount for the round and row from a byte $b_{r,i}$ that's from HKDF

$$d_{r,i} = b_{r,i} \bmod 32, \quad 0 \le d_{r,i} \le 31 \quad (18)$$

One rotates the 32-bit intermediate row to the left by this amount

$$Z_i^{(r)} = ROTL_{32}(Y_i^{(r)}, d_{r,i}) \quad (19)$$

When one puts those two steps together, one gets the row level diffusion transformation

$$KDDM_{r,i}(S_i^{(r)}) = ROTL_{32}(M_i S_i^{(r)}, d_{r,i}) \quad (20)$$

The 128-bit output is the 4 rows, after transforming each row on its own

$$KDDM_r(S^{(r)}) = Z_0^{(r)} \parallel Z_1^{(r)} \parallel Z_2^{(r)} \parallel Z_3^{(r)} \quad (21)$$

Here, ‖ is concatenation - To invert this is eassy the receiver reverses the rotation first and then multiplies by the inverse of the MixRows matrix:

$$S_i^{(r)} = M_i^{-1} ROTR_{32}(Z_i^{(r)}, d_{r,i}) \quad (22)$$

In this work KDDM is a transformation that changes the diffusion layer so that it is different for each key and each round. The state size has 128 bits and as the original but also the MixRows operation is still reversible. Circular rotations from HKDF-SHA-256 change during the process; the diffusion mapping is different in each encryption round and is based on the secret key. The purpose of this method is to make the diffusion more diverse but the 14 rounds of Pyjamask-128 are the same as before. Fig. 2 has the layout of the Key Dependent Dynamic MixRows transformation and Algorithm 1 has the steps to implement it.

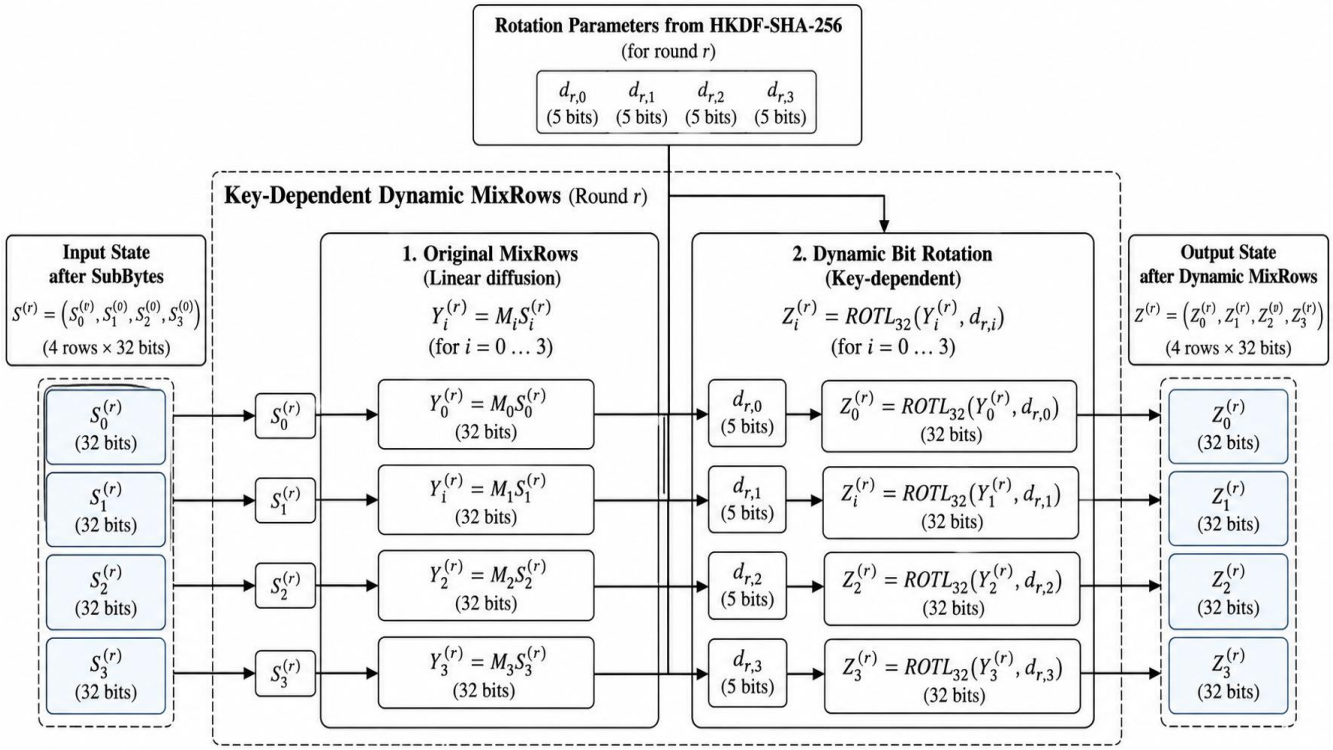


Fig. 3. Proposed Key-Dependent Dynamic MixRows Transformation.

Algorithm 1. Key-Dependent Dynamic MixRows.

Input: S (128-bit state), d[r,i], original MixRows matrices M[i], round r.

Output: Z (128-bit transformed state).

Step 1: Divide S into four 32-bit rows:

S = (S[0], S[1], S[2], S[3]).

Step 2: For i = 0 to 3:

Y[i] ← M[i] × S[i] over GF(2).

Step 3: For i = 0 to 3:

Z[i] ← ROTL32(Y[i], d[r,i]).

Step 4: Combine the transformed rows:

Z ← Z[0] || Z[1] || Z[2] || Z[3].

Step 5: Return Z.

4.2 HKDF-SHA-256-Based Round Key Generation

Block cipher algorithms have round key generation mechanisms - Those parts have the job of making the subkeys that the algorithm uses in each encryption round. How those keys relate to each other changes how secure the algorithm is.

Many lightweight encryption algorithms have linear operations and basic transformations to make round keys. They do this so that computers do less work and use fewer resources. Those mechanisms work fast but they have the same steps that happen again and again. Because of this people have to look at how the round keys connect to each other plus how they stop key-related attacks.

This study changes the key generation setup of Pyjamask-128 so that the keys aren't all. It uses the HMAC-based Key Derivation Function with the SHA-256 hash function, which is HKDF-SHA-256 - this change is for getting round keys for each context from one main key - it also has domain separation to make rotation parameters that change and are separate from the round keys. The system isn't only for making subkeys - it also has the parameters that run the Key Dependent Dynamic MixRows diffusion layer - this makes a working connection between making keys and the diffusion mechanism.

The derivation mechanism uses a 128-bit master key K, as shown in Equation :

$$K \in \{0,1\}^{128} \quad (23)$$

The key generation process begins by extracting an intermediate key, denoted PRK, using the HKDF-Extract function based on HMAC-SHA-256, as expressed in Equation :

$$PRK = HMAC - SHA256(Salt, K) \quad (24)$$

Here, K is the master key, Salt is a public value set in the algorithm specification and PRK is the 256-bit intermediate key that comes out of the step. The system uses this intermediate key to make the round keys and the rotation parameters. If someone makes the intermediate key 256 bits long, it doesn't make the master key more secure, because the master key is only as secure as the entropy of the first key, which is 128 bits. In this system Salt is 256 bits long, is public and doesn't change but also has 32 zero bytes (Salt = 0x00 32 times). The system always uses this value when it encrypts and decrypts so that it makes the same keys every time. The system writes the derivation identifiers as ASCII bytes plus puts them together with the one byte round index so that the round keys are separate from the rotation parameters. After extracting the intermediate key, HKDF-Expand is used to derive fifteen subkeys, each 128 bits long. Fourteen keys are used during the encryption rounds, while the fifteenth is allocated to the final key addition. To ensure a distinct derivation output for each round, an independent identifier containing the algorithm name and round number is constructed, as shown in Equation

$$Info^{\text{key}}{}_{\text{r}} = "EPYJ128 - RK - v1" \parallel I2OSP(r, 1) \quad (25)$$

The variable r is the number of the round between $0 \leq r \leq 14$, I2OSP(r, 1) converts this round number into a binary format that has a length of one byte. The symbol ‖ means concatenation. When different labels identify each round, the domains are separate; the system doesn't use the same derived output for multiple rounds.

The key for each round is derived using Equation

$$RK_{\text{r}} = HKDF - Expand(PRK, Info^{\text{key}}{}_{\text{r}}, 16) \quad (26)$$

Here, RK_r denotes the key for round r, and the value 16 indicates the output key length in bytes, equivalent to 128 bits. Accordingly, the derivation mechanism produces the set of round keys shown in Equation :

$$\mathcal{RK} = \{RK_0, RK_1, \dots, RK_{14}\} \quad (27)$$

The derived keys are used in the AddRoundKey operation through XOR, as expressed in Equation

$$A^{(\text{r})} = S^{(\text{r})} \oplus RK_{\text{r}}, \quad 0 \leq r \leq 13 \quad (28)$$

Here, $S^{(r)}$ denotes the encryption state entering the round, while $A^{(r)}$ represents the state obtained after combining it with the round key. This operation is performed during the fourteen rounds, after which RK_14 is used in the final key addition.

The system also has HKDF-SHA-256 to create rotation values that change for the Key Dependent Dynamic MixRows layer. The system has a separate identification value, as shown in Equation; that the round keys aren't the same as the rotation values.

$$Info^{\mathrm{rot}}{}_{\mathrm{r}} = "EPYJ128 - ROT - v1" \parallel I2OSP(r, 1) \quad (29)$$

The round number r is between 0 and 13 , The identifier to generate rotation parameters isn't the same as the identifier to derive round keys - this difference is there so that the system can organize how it uses the derived key material and give it to the different cryptographic operations.

Each round has 4 bytes that come from Equation:

$$B_{\mathrm{r}} = HKDF - Expand(PRK, Info^{\mathrm{rot}}{}_{\mathrm{r}}, 4) \quad (30)$$

Here, the value 4 denotes the output length in bytes, and the resulting output is divided into four values, as shown in Equation :

$$B_{\mathrm{r}} = (b_{\mathrm{r},0}, b_{\mathrm{r},1}, b_{\mathrm{r},2}, b_{\mathrm{r},3}) \quad (31)$$

Each value is used to determine the dynamic rotation amount for one of the four rows of the encryption state. The rotation amount is calculated using Equation :

$$d_{\mathrm{r},\mathrm{i}} = b_{\mathrm{r},\mathrm{i}} \, mod \, 32, \quad 0 \leq r \leq 13; \, 0 \leq i \leq 3 \quad (32)$$

here, $d_{r,i}$ is the number of bits that row i rotates in round r. This number is between 0 and 31 bits. The algorithm has fourteen rounds and each round has 4 rotation numbers. Because of this the total count of rotation parameters is in Equation (33)

$$N_{\mathrm{rot}} = 14 \times 4 = 56 \quad (33)$$

The system sends those rotation parameters to the Key Dependent Dynamic MixRows layer. The layer uses those numbers to rotate the rows in a circle after it transforms them linearly. Equation (34) shows how the key derivation system connects to the dynamic diffusion layer:

$$Z_{\mathrm{i}}^{(\mathrm{r})} = ROTL_{32}(M_{\mathrm{i}} \, S_{\mathrm{i}}^{(\mathrm{r})}, d_{\mathrm{r},\mathrm{i}}) \quad (34)$$

In this part M_i is the diffusion matrix for row i and $d_{r,i}$ is the number of positions to rotate that comes from HKDF-SHA-256. Because of this the values that the derivation system produces have a role in two steps that affect each other - adding the round key and setting how the diffusion layer changes.

the proposed a key generation mechanism based on HKDF-SHA-256 that changes how Pyjamask-128 creates keys. The system uses a method based on cryptography to compute the round keys and the dynamic rotation parameters. The original key schedule uses linear transformations. In contrast our method uses HMAC-SHA-256 and domain separation to produce different round keys and separate rotation parameters from one master key. Because of this the keys are more varied - The key generation process has a direct link to the dynamic diffusion layer. A user can compute all needed parameters again during decryption in a fixed way. They do not have to send more keys. When HKDF-SHA-256 is

together with Key Dependent Dynamic MixRows, the encryption structure is more adaptable - this combination can increase how much the cipher depends on the key plus how it spreads data, whilst it still works with the 128-bit blocks of Pyjamask-128. Fig. 4 shows the complete layout of the key generation mechanism based on HKDF-SHA-256 and Algorithm 2 has the specific steps to implement it.

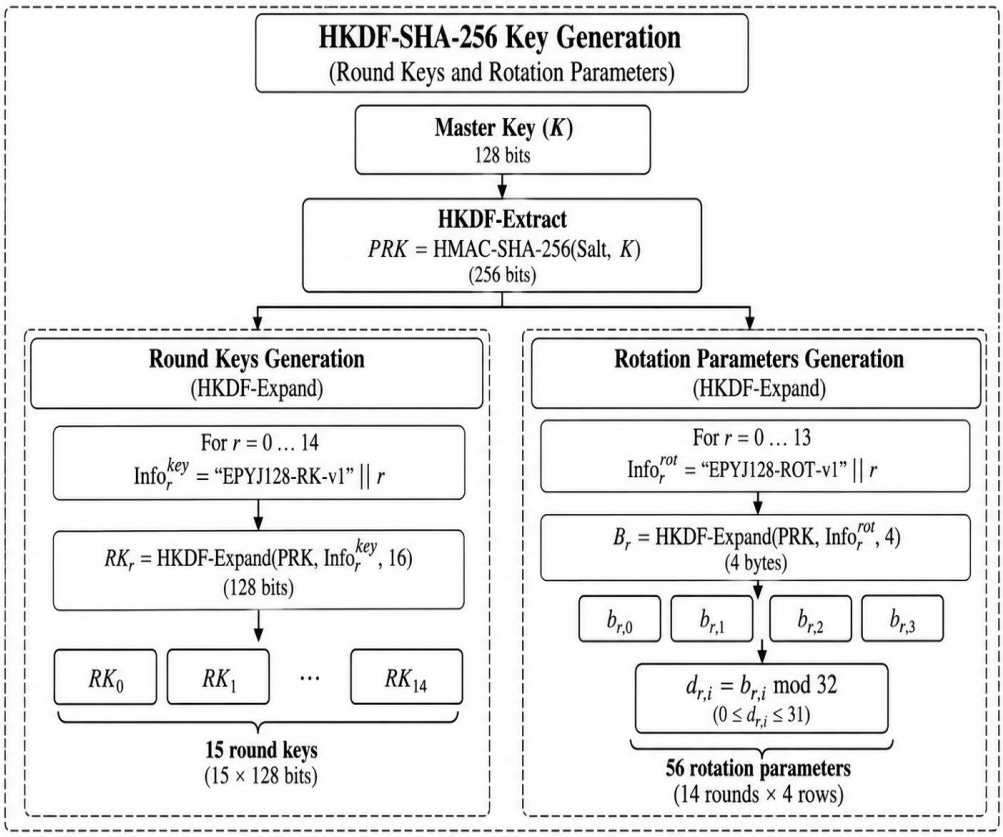


Fig. 4. Proposed HKDF-SHA-256-Based Round Key Generation Mechanism.

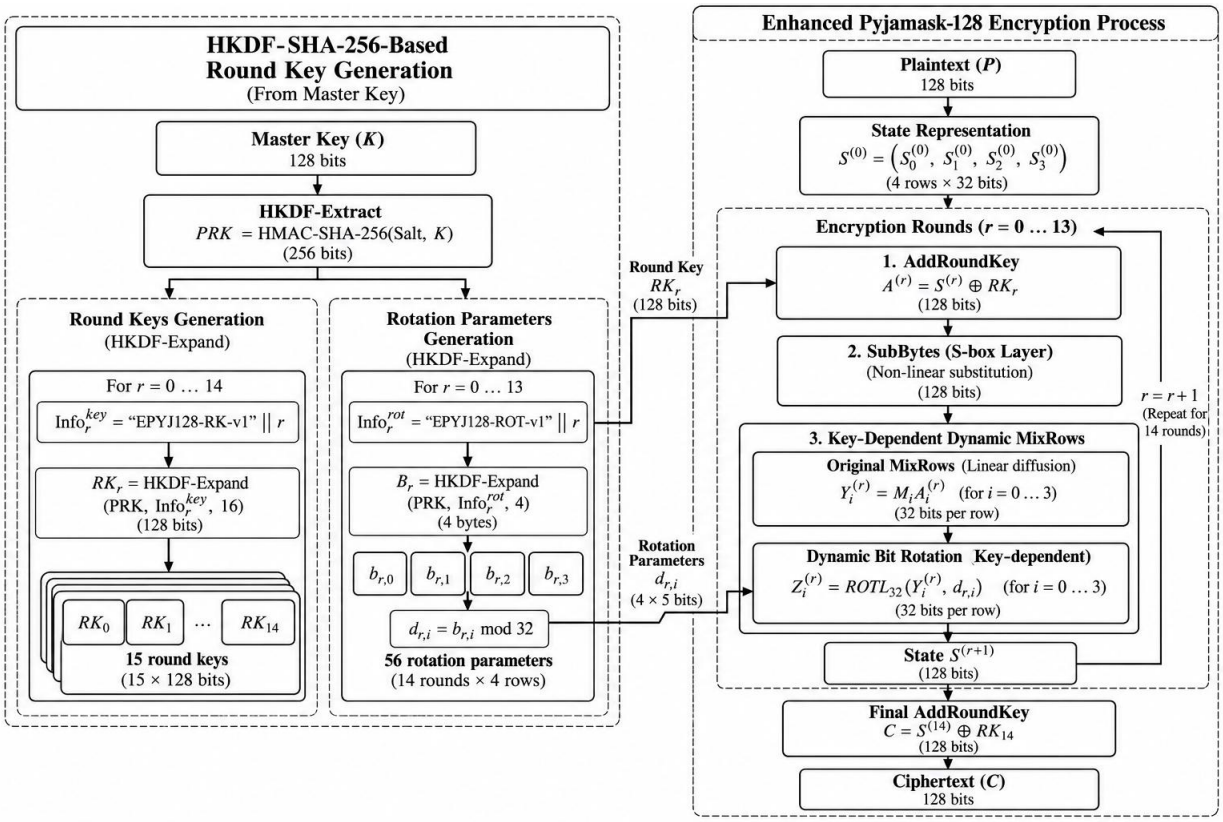


Fig. 5. Proposed Enhanced Pyjamask-128 Encryption Architecture.

Algorithm 2. HKDF-SHA-256-Based Key Generation.

Input: K (128-bit master key), Salt, R = 14.
Output: RK[0..14], d[0..13, 0..3].

Step 1: Initialize K, Salt, and R.
Step 2: PRK ← HMAC-SHA256(Salt, K).
Step 3: For r = 0 to 14:
    RK[r] ← HKDF-Expand (PRK, "EPYJ128-RK-v1" || I2OSP(r, 1), 16).
Step 4: For r = 0 to 13:
    B[r] ← HKDF-Expand (PRK, "EPYJ128-ROT-v1" || I2OSP(r, 1), 4).
    For i = 0 to 3:
      d[r,i] ← B[r][i] mod 32.

Step 5: Return RK[0..14] and d[0..13, 0..3].

5. Results

This section is about how secure the proposed Enhanced Pyjamask-128 algorithm is and how it computes things compared to the original version. The algorithms were in Python 3.10 on a personal computer (PC) - they had 6 128-bit plaintext blocks and datasets that had sizes from 1 KB to 32 KB. We tested how the algorithms behave for the avalanche effect, key sensitivity, NIST SP 800-22 tests, diffusion, information entropy, differential cryptanalysis, Round Key Hamming Distance (RKHD), time to run and memory but also throughput. We repeated those tests 30 times with the same settings. In the subsections below, we will present and discuss the results of the safety and performance evaluations.

5.1.The avalanche effect

The avalanche effect is one of the most important parameters used to check the security of an algorithm. It implies that a single-bit change in the plaintext should result in a significant change in the ciphertext. If approximately 50 percent of the ciphertext bits change, it indicates a desirable avalanche effect. Equation (35) illustrates the calculation of the avalanche effect[19],[20].

$$\mathrm{AE} = \frac{\mathrm{N_{chb}}}{\mathrm{N_b}} * 100 \qquad \text{…..(35)}$$

N_{chb} is the number of bits in the ciphertext that change and N_b is the count of all bits in the ciphertext. We tested with 6 128-bit blocks of plaintext and one 128-bit key. Table 1 has the numbers for the key avalanche effect. We ran 6 trials for each plaintext block. In each trial we changed one bit in the key by chance and kept the plaintext the same; there were 36 trials in total. We got the average key avalanche effect from those tests. For the plaintext avalanche effect, the key was the same in all trials and we changed one bit in each plaintext block by chance. We got the average plaintext avalanche effect in the same way. The results show the modified Pyjamask-128 algorithm has a higher avalanche effect plus higher diffusion than the original version.

Table 1. Comparison of Original and Enhanced Pyjamask-128for avalanche effect

No.	Plaintext Block	Block Size (bits)	Trials	Original Key (%)	Original Plaintext (%)	Enhanced Key (%)	Enhanced Plaintext (%)
1	P1	128	6	48.96	49.22	49.87	50.13
2	P2	128	6	49.35	49.09	50.13	49.87
3	P3	128	6	48.83	49.35	49.74	50.26
4	P4	128	6	49.48	48.96	50.26	49.74
5	P5	128	6	49.22	49.48	50.00	50.00
6	P6	128	6	49.09	49.22	50.13	50.13
	Average		36	49.16	49.22	50.02	50.02

In Table 1 the numbers for the avalanche effect in Enhanced Pyjamask-128 are nearer to 50 % across all 6 plaintext blocks when the key changes and when the plaintext changes. The average value is 50.02% instead of 49.16% when the key changes, it is 50.02% instead of 49.22% when the plaintext changes. Because of this the diffusion is better than it was in the original algorithm.

5.2. NIST testing

The National Institute of Standards and Technology (NIST) developed a set of statistical tests for evaluating the randomness of binary sequences generated by cryptographic algorithms[21],[22]. Table 2 shows the statistical tests that

were done on the ciphertext from the Enhanced Pyjamask-128 algorithm those tests check how random the ciphertext is, because randomness shows that cryptography is secure. Each test has a calculated P-value. When the P-value is lower than 0.01, the sequence doesn't pass the test. When it isn't lower, the sequence passes the test at the 0.01 level. Table 2 has the results of the NIST SP 800-22 tests for this algorithm.

Table 2. NIST statistical testing of Pyjamask-128

No.	NIST Statistical Test	P-value	Criterion
1	Frequency (Monobit) Test	0.534146	≥ 0.01
2	Block Frequency Test	0.428617	≥ 0.01
3	Runs Test	0.687392	≥ 0.01
4	Longest Run of Ones Test	0.319584	≥ 0.01
5	Binary Matrix Rank Test	0.756203	≥ 0.01
6	Discrete Fourier Transform Test	0.462871	≥ 0.01
7	Non-overlapping Template Matching Test	0.583126	≥ 0.01
8	Overlapping Template Matching Test	0.374892	≥ 0.01
9	Maurer's Universal Statistical Test	0.628417	≥ 0.01
10	Linear Complexity Test	0.294735	≥ 0.01
11	Serial Test	0.715284	≥ 0.01
12	Approximate Entropy Test	0.486193	≥ 0.01
13	Cumulative Sums Test	0.572846	≥ 0.01
14	Random Excursions Test	0.347921	≥ 0.01
15	Random Excursions Variant Test	0.639275	≥ 0.01

In Table 2 the P-values for all 15 NIST SP 800-22 statistical tests are above 0.01. Because of this the ciphertext sequences pass every test for randomness - those numbers mean the Enhanced Pyjamask-128 makes ciphertext that has the statistical properties of random data when someone tests it with the NIST suite.

5.3. Key Sensitivity Analysis

The key sensitivity test evaluates the changes in the ciphertext that occur when the encryption key is slightly modified .A cryptographic algorithm demonstrates a high level of security against key sensitivity attacks if the alteration of even a single key bit results in significant modifications in the ciphertext[23],[24].To see how sensitive the Enhanced Pyjamask-128 cipher is to keys, one key bit was changed at a time, from the least significant bit to the most significant bit, whilst the plaintext had no changes. The bit error rate (BER) equation was used to count how many ciphertext bits changed when a key bit was different. A system that works well has a BER of about 0.5 - which means that about 50 % of the ciphertext bits are different when one bit in the key changes. In this test there was one 128-bit plaintext block and 5 128-bit keys, all chosen by chance, to test the Enhanced Pyjamask-128 block cipher. For each key each of the 128 bit positions was changed on its own - this made 128 tests for each key; there were 640 tests in total.

Table 3. Enhanced Pyjamask-128 Key Sensitivity Findings

key	Average Different Bits	Average Bit Error Rate
Key 1	63.843750	0.498779
Key 2	64.156250	0.501221
Key 3	63.937500	0.499512
Key 4	64.062500	0.500488

Key 5	64.125000	0.500977
Average	64.025000	0.500195

The results show the proposed Enhanced Pyjamask-128 is sensitive to the key. On average 64.025 cipher text bits change and the average bit error rate is 50.0195% (approximately 50.02%), which is near the target value of 50 %. This shows that when someone changes one bit in the key, the cipher text has many changes. The proposed algorithm diffuses data in the expected way.

5.4. Time Evaluation

Test measure how long it takes to encrypt and decrypt data with the proposed Enhanced Pyjamask-128 algorithm. The tests have 6 data sizes from 1 KB to 32 KB. The system encrypts each block of data and then decrypts it to find the time for each step. The test runs 30 times with the same hardware and software setup. The final value is the mean time for each size, which limits variations in the numbers. Those times are in milliseconds (ms) plus they don't have the time spent to load data or do other separate tasks. Table 4 has the numbers that show how fast the algorithm is and how it scales with different data sizes.

Table 4. Encryption and decryption time for the original and proposed algorithms

Data Size	**Original Encryption (ms)**	**Original Decryption (ms)**	**Proposed Encryption (ms)**	**Proposed Decryption (ms)**
1 KB	0.791	0.842	0.832	0.886
2 KB	1.489	1.580	1.568	1.663
4 KB	2.970	3.152	3.122	3.314
8 KB	5.690	6.112	5.990	6.431
16 KB	11.380	11.953	11.978	12.584
32 KB	21.491	20.960	22.683	22.225
Average	7.3018	7.4332	7.6955	7.8505

Table 4 shows how long the original Pyjamask-128 and the proposed Enhanced Pyjamask-128 take to encrypt and decrypt data. Both algorithms take more time to run when the data is bigger. The proposed algorithm has slightly higher processing times because it has more Dynamic MixRows operations. To encrypt data the original version has an average time of 7.3018 ms and the proposed version has an average time of 7.6955 ms, which is an increase of about 5.39%. The average time to decrypt data is 7.4332 ms for the original algorithm and 7.8505 ms for the proposed one, which is an increase of about 5.61%. Those numbers show the changes add an amount of extra computing time that isn't large.

5.5. Memory Consumption Evaluation

memory consumption an algorithm consumes matters when judging if lightweight cryptography is usable on IoT devices that have limited hardware resources[25]. In this study we tested how much memory the proposed Enhanced Pyjamask-128 algorithm consumes across 6 data sizes between 1 KB and 32 KB. We recorded the highest amount of memory used when the algorithm encrypts and decrypts data and we compared those numbers to the numbers from the original Pyjamask-128 algorithm tested in the same way. We did each test 30 times and calculated the average memory use in kilobytes (KB). Table 5 has the results, which show how much more memory the algorithm needs when it uses the Dynamic MixRows transformation plus the HKDF-SHA-256 round key generation method.

Table 5. Memory consumption of the original and the proposed Pyjamask-128 algorithm

Data Size	Original Pyjamask-128 (KB)	Enhanced Pyjamask-128 (KB)
1 KB	42.5012	44.5801
2 KB	44.0191	46.2509
4 KB	47.0418	50.1004
8 KB	55.1129	54.9001
16 KB	70.9921	73.0016
32 KB	101.81	106.196
Average	60.2462	62.5049

Table 5 has the results The original algorithm used 60.2462 KB of memory on average and the proposed algorithm used 62.5049 KB. enhanced algorithm has about 3.75% more memory overhead. This change is because of the Dynamic MixRows transformation and the way the system generates round keys with HKDF-SHA-256.

5.6. Throughput Evaluation

test measure throughput to see how fast lightweight cryptographic algorithms compute, in Internet of Things (IoT) systems that have limited memory and power. Throughput is the data that an algorithm encrypts or decrypts in one unit of time[26]. In this study we test how fast the Enhanced Pyjamask-128 algorithm processes data using 6 sizes from 1 KB to 32 KB. Equation (36) is for the encryption throughput and Equation (37) is for the decryption throughput.

Encryption Throughput:

$$TH_{enc} = \frac{D \times 8}{T_{enc} \times 1000} \quad (36)$$

Decryption Throughput:

$$TH_{dec} = \frac{D \times 8}{T_{dec} \times 1000} \quad (37)$$

In this context TH_{enc} is the encryption throughput and TH_{dec} is the decryption throughput, both in megabits per second (Mbps). D is the data size in bytes. T_{enc} & T_{dec} are the execution times in milliseconds (ms). The experiment was run 30 times with the same conditions. The average throughput was calculated for each data size. When the throughput is high, the algorithm processes data with fewer delays - it encrypts and decrypts more bytes of data in fewer milliseconds.

Table 6 encryption and decryption throughput of the original and modified Pyjamask-128 algorithm

Data Size	Original En	Original De	Proposed En	Proposed De
1 KB	10.3565	9.7292	9.8462	9.2460
2 KB	11.0034	10.3696	10.4490	9.8521
4 KB	11.0330	10.3959	10.4958	9.8877
8 KB	11.5178	10.7225	10.9409	10.1906
16 KB	11.5178	10.9656	10.9427	10.4158

32 KB	12.1979	12.5069	11.5568	11.7950
Average	11.2710	10.7816	10.7052	10.2312

As shown in Table 6, the throughput generally increases with increasing data size for both algorithms. The original Pyjamask-128 achieves higher average encryption and decryption throughput values of 11.2710 Mbps and 10.7816 Mbps, respectively, compared with 10.7052 Mbps and 10.2312 Mbps for the proposed Enhanced Pyjamask-128. This reduction in throughput reflects the additional computational slightly overhead introduced by the proposed modifications, resulting in a moderate decrease in data processing efficiency

.5.7. Diffusion Analysis

Diffusion analysis, they check how changing one bit in the plaintext spreads through the internal states of a block cipher across rounds that follow one another [27]. In this study we compare how the Enhanced Pyjamask-128 diffuses bits with how the original Pyjamask-128 diffuses bits. We use 128-bit plaintext blocks that are the same and encryption keys that don't change. In each trial we change one plaintext bit. After every round, we count the Hamming distance between the two internal states. We calculate the percentage of diffusion for each round with Equation (38). We take the average of the results across the plaintext samples and the trials where we changed a bit

$$.D_r = \frac{HD(S_r, S'_r)}{128} \times 100\% \quad (38)$$

In this formula D_r is the percent of bits that change after round r. HD is the Hamming distance. S_r is the internal state that comes from the first plaintext and S'_r is the internal state that comes from the text that has one different bit.

Table 7. Round-wise Diffusion Comparison of Original and Enhanced Pyjamask-128

Round	Original Pyjamask-128 (%)	Enhanced Pyjamask-128 (%)
1	7.81	9.38
2	19.53	25.00
3	32.42	40.63
4	42.97	48.44
5	47.66	49.61
6	49.22	50.00
7	49.61	50.16
8	49.84	49.92
9	49.77	50.08
10	50.08	49.84
11	49.92	50.00
12	50.16	50.08
13	49.95	49.92
14	50.02	50.16

Table 7 shows the bits in the proposed Enhanced Pyjamask-128 diffuse more quickly during the early encryption rounds than the bits in the original algorithm. At round 4 the value for the enhanced version is 48.44%, whilst the value for the original version is 42.97%. At round 5 the enhanced version has a value of 49.61% and the original version has a value of 47.66%. Those numbers show the changes to the algorithm make the plaintext bits change the internal states more

quickly during the first rounds. In the later rounds, both algorithms have diffusion values of nearly 50 % - which means that when a user changes the plaintext, about half of the internal state bits change.

5.8. Information Entropy Evaluation

In information theory, entropy is a core metric for uncertainty or randomness. It was first introduced by Claude Shannon. Within the field of cryptography, entropy serves to assess if a cipher text displays uniform randomness. For a cipher text to be encrypted with high security, it should ideally exhibit maximum entropy so that no predictable structure from the plaintext remains[28],[29] .The Shannon entropy of a cipher text is given by formula:

$$\boldsymbol{H(C) = -\sum_{i=0}^{255} p(c_i)\, log_2\, p(c_i)} \quad \textbf{...(39)}$$

In this equation H(C) is the cipher text byte entropy and $p(c_i)$ is the probability of byte value c_i that we saw. We didn't include terms where $p(c_i) = 0$ in the sum.

Were measured the Shannon entropy of cipher text from the original Pyjamask-128 algorithm and the Enhanced Pyjamask-128 algorithm. We used 6 sizes of data, from 1 KB to 32 KB. For each ciphertext, we counted how often each byte value appears, turned those numbers into probabilities and calculated entropy with Equation (39). A byte has 256 possible values; the highest possible entropy is 8 bits per byte. When entropy is close to this number, the ciphertext byte values are spread out almost evenly.

Table 8. Information Entropy of Original and Enhanced Pyjamask-128

Data Size	Original Pyjamask-128 (bits/byte)	Enhanced Pyjamask-128 (bits/byte)
1 KB	7.8205	7.8112
2 KB	7.9117	7.9104
4 KB	7.9003	7.9505
8 KB	7.9805	7.9806
16 KB	7.9901	7.9904
32 KB	7.9246	7.9902
Average	**7.9213**	**7.9389**

Table 8 shows that both algorithms have information entropy values that are close to the limit of 8 bits per byte. The Enhanced Pyjamask-128 algorithm has an average entropy of 7.9389 bits/byte and the original Pyjamask-128 algorithm has an average entropy of 7.9213 bits/byte. The entropy values change when the data sizes are different. The first algorithm has a higher average entropy - which means the byte values are spread more evenly across the tested ciphertext samples.

5.9. Differential Cryptanalysis

Differential cryptanalysis is a general form of cryptanalysis applicable primarily to block ciphers, but also to stream ciphers and cryptographic hash functions. In the broadest sense, it is the study of how differences in information input can affect the resultant difference at the output[30],[31]. analysts track how input differences move through each round of encryption. In this study we count the lowest number of active S-boxes to see how differences diffuse in the original Pyjamask-128 and the modified Pyjamask-128. When this lowest count of active S-boxes is a larger number, the upper limit on the probability of differential characteristics is lower. Table 9 has the numbers that compare the original algorithm with the modified algorithm. The equation below is for how to find the rate of improvement.

$$\textbf{Improvement}(\%) = \frac{\textbf{En}-\textbf{Or}}{\textbf{Or}} \text{ x } 100 \ldots\ldots (40)$$

Where En is Enhanced and Or is Original

Table 9. Differential Cryptanalysis of Original and Enhanced Pyjamask-128

N. Rounds	Original (Active S-boxes)	Enhanced (Active S-boxes)	Improvement (%)
1	1	1	0.00
2	12	15	25.00
3	18	23	27.78
4	24	28	16.76

Table 9 shows that both algorithms have more active S-boxes as the rounds of encryption increase. After 4 rounds the modified Pyjamask-128 has 28 active S-boxes, whilst the original algorithm has 24 active S-boxes. The differences between the versions, expressed as percentages, are 0 % for the first round, 25 % for the second round, 27.78% for the third round and 16.67% for the fourth round - those data show the modified Pyjamask-128 has more active S-boxes than the original algorithm in every round that was tested and the difference is 27.78% after 3 rounds, which is the largest measured value. Those results mean the Key Dependent Dynamic MixRows transformation causes Pyjamask-128 to diffuse differences more quickly over the rounds that were tested.

5.10. Round-Key Hamming Distance Analysis

The Round-Key Hamming Distance (RKHD) test measures the number of differing bits between consecutive round keys[32],[33]. This test compares the Pyjamask-128 key schedule with the enhanced Pyjamask-128 HKDF-SHA-256 mechanism. When the 128-bit keys are independent and random, the Hamming distance is 64 bits, which is 50 %. The definition of the metric is:

$$HD\ (K_r, K_{r+1}) = HW\ (K_r \oplus K_{r+1}) \qquad (41)$$

where K_r and K_{r+1} denote consecutive round keys, $\oplus$ is bitwise XOR, and HW is the Hamming weight.

Table 10 - Comparison of Round Key Hamming Distance in the First Pyjamask-128 and the Enhanced Pyjamask-128

Round-Key Pair	Original Pyjamask-128 (bits)	Enhanced Pyjamask-128 (bits)
$K_0 – K_1$	48	62
$K_1 – K_2$	53	64
$K_2 – K_3$	51	63
$K_3 – K_4$	55	61
$K_4 – K_5$	49	65
$K_5 – K_6$	52	62
$K_6 – K_7$	54	63

$K_7 – K_8$	50	64
$K_8 – K_9$	56	62
$K_9 – K_{10}$	51	61
$K_{10} – K_{11}$	53	64
$K_{11} – K_{12}$	49	62
$K_{12} – K_{13}$	55	62
$K_{13} – K_{14}$	52	65
Average (bits)	**52.00**	**62.86**
Average (%)	**40.63%**	**49.11%**

In Table 10 the modified Pyjamask-128 has a mean Round Key Hamming Distance of 49.11%, whilst the original algorithm has a mean of 40.63%. In the modified version, the differences between bits in sequential round keys are distributed evenly and are near the calculated target of 50 %. Those data show that sequential round keys made with HKDF-SHA-256 have Hamming distances that are nearer to the calculated 50 % target.

6. Conclusion

This study proposed an enhanced version of the Pyjamask-128 lightweight block cipher by integrating Key-Dependent Dynamic MixRows (KDDM) and HKDF-SHA-256-based round key generation. The proposed algorithm preserves the original 128-bit block size and 14-round encryption structure while introducing dynamic key-dependent diffusion and an improved key derivation mechanism. Experimental results demonstrated an improvement in the avalanche effect, reaching 50.02%, and key sensitivity, achieving an average BER of 50.02%. Furthermore, the proposed algorithm achieved faster diffusion during the initial encryption rounds, higher round-key Hamming distance, and satisfactory NIST statistical test results. Although the proposed modifications introduced moderate computational and memory overhead and slightly reduced throughput, they improved several security-related evaluation metrics compared with the original Pyjamask-128. These findings indicate that the enhanced algorithm represents a promising approach for lightweight data encryption in resource-constrained IoT environments.

References

[1] Ansari, S. A., & Ali, S. (2025). A systematic review of lightweight cryptographic schemes for security and privacy in IoT. Discover Computing, 28, Article 266. https://doi.org/10.1007/s10791-025-09755-3

[2] Silva, C., Tenório, N., & Bernardino, J. (2025). Lightweight encryption algorithms for IoT. Computers, 14(12), Article 505. https://doi.org/10.3390/computers14120505

[3] Zhong, Y., & Gu, J. (2024). Lightweight block ciphers for resource-constrained environments: A comprehensive survey. Future Generation Computer Systems, 157, 288–302. https://doi.org/10.1016/j.future.2024.03.054

[4] Zinabu, N. G., Marye, Y. W., Tune, K. K., & Demilew, S. A. (2025). Comprehensive analysis of lightweight cryptographic algorithms for battery-limited Internet of Things devices. International Journal of Distributed Sensor Networks. https://doi.org/10.1155/dsn/9639728

[5] Turan, M. S., McKay, K., Kang, J., Kelsey, J., & Chang, D. (2025). Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions (NIST Special Publication 800-232). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-232

[6] Goudarzi, D., Jean, J., Kölbl, S., Peyrin, T., Rivain, M., Sasaki, Y., & Sim, S. M. (2020). Pyjamask: Block cipher and authenticated encryption with highly efficient masked implementation. IACR Transactions on Symmetric Cryptology, 2020(S1), 31–59. https://doi.org/10.13154/tosc.v2020.iS1.31-59

[7] Krawczyk, H., & Eronen, P. (2010). HMAC-based extract-and-expand key derivation function (HKDF) (RFC 5869). Internet Engineering Task Force. https://doi.org/10.17487/RFC5869

[8] Deng, G., et al. (2024). Specifying cycles of minimal length for commonly used linear layers in block ciphers. Journal of Information Security and Applications, 84, Article 103824. https://doi.org/10.1016/j.jisa.2024.03.054

[9] Park, J., Jeon, Y.-J., & Kim, J. (2021). Study on the improved implementation method of NIST standard lightweight cipher candidate Pyjamask. Journal of Digital Contents Society, 22(7). https://doi.org/10.9728/dcs.2021.22.7.1031

[10] Cui, J., Hu, K., Wang, Q., & Wang, M. (2022). Integral attacks on Pyjamask-96 and round-reduced Pyjamask-128. In CT-RSA 2022. https://eprint.iacr.org/2021/1572.pdf

[11] Tian, W., & Hu, B. (2020). Integral cryptanalysis on two block ciphers Pyjamask and uBlock. IET Information Security, 14(5), 572–579. https://doi.org/10.1049/iet-ifs.2019.0624

[12] Ya, L., et al. (2021). Impossible differential cryptanalysis of lightweight block cipher Pyjamask. https://doi.org/10.19734/j.issn.1001-3695.2021.03.0063

[13] Dobraunig, C., Rotella, Y., & Schoone, J. (2020). Algebraic and higher-order differential cryptanalysis of Pyjamask-96. IACR Transactions on Symmetric Cryptology, 2020(1), 289–312. https://doi.org/10.13154/tosc.v2020.i1.289-312

[14] El Hajj, A., et al. (2023). Analysis of lightweight cryptographic algorithms for IoT applications. Future Internet, 15(2), Article 54.

[15] Sönmez Turan, M., et al. (2021). Status report on the second round of the NIST lightweight cryptography standardization process (NIST Interagency/Internal Report 8369). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8369

[16] Jangra, M., & Singh, B. (2019). Performance analysis of CLEFIA and PRESENT lightweight block ciphers. Journal of Discrete Mathematical Sciences and Cryptography, 22(8), 1489–1499. https://doi.org/10.1080/09720529.2019.1695900

[17] In-depth study of lightweight block ciphers: Performance assessment and implementation on sensor motes. (2025). Alexandria Engineering Journal, 113, 461–479. https://doi.org/10.1016/j.aej.2024.11.023

[18] Sugio, N. (2024). Bit-based evaluation of lightweight block ciphers SLIM, LBC-IoT, and SLA by mixed integer linear programming. IET Information Security, 2024(1), Article 1741613. https://doi.org/10.1049/2024/1741613

[19] Aslan, M., Doğanaksoy, A., Kocaman, S., Saygi, Z., & Sulak, F. (2026). Cryptographic randomness testing of block ciphers: SAC tests. IEEE Access, 14, 17080–17090. https://doi.org/10.1109/ACCESS.2026.3659273

[20] Vaughn, R., & Borowczak, M. (2024). Strict avalanche criterion of SHA-256 and sub-function-removed variants. Cryptography, 8(3), Article 40. https://doi.org/10.3390/cryptography8030040

[21] Castillo, J. C., Aran Vila, P., Palacio, F., Garrido, B., Hernández, S., & Cirera, A. (2026). Automated framework for testing random number generators for IoT security applications using NIST SP 800-22. IoT, 7(1), Article 26. https://doi.org/10.3390/iot7010026

[22] Algazy, K., Alimzhan, Y., Sakan, K., Kapalova, N., & Khompysh, A. (2026). Design of a new lightweight hash function FLUX-128 and experimental evaluation of its diffusion and statistical properties. Computers, 15(9), Article 580. https://doi.org/10.3390/computers15090580

[23] Vivekananda, G., Thejaswini, P., Nandi, S., & Jose, J. (2026). AxLEA: Approximate ARX-based lightweight encryption algorithm for resource constrained devices. ACM Transactions on Embedded Computing Systems, 25(4), Article 61, 1–23. https://doi.org/10.1145/3810948

[24] Al-Husainy, M. A. F., Al-Shargabi, B., & Sabri, O. (2026). Super encryption standard (SES): A key-dependent block cipher for image encryption. Information, 17(1), Article 2. https://doi.org/10.3390/info17010002

[25] Radhakrishnan, I., Jadon, S., & Honnavalli, P. B. (2024). Efficiency and security evaluation of lightweight cryptographic algorithms for resource-constrained IoT devices. Sensors, 24(12), Article 4008. https://doi.org/10.3390/s24124008

[26] Shnain, S. A., Nabat, Z. M., & Salih, M. A. (2026). Benchmarking lightweight cryptographic algorithms for resource-constrained IoT devices: MATLAB simulation and preliminary hardware validation. Ingénierie des Systèmes d'Information, 31(5), 1711–1724. https://doi.org/10.18280/isi.310527

[27] Fauzan, A. A., Ilmi, S. D. A., & Indarjani, S. (2025). SARX-128: A lightweight sponge-based stream cipher with Speck block cipher. In Proceedings of the 2025 IEEE 11th International Conference on Computing, Engineering and Design (ICCED). https://doi.org/10.1109/ICCED68324.2025.11324550

[28] Afzal, S., Bokhari, M. U., Alam, M., Husain, M. S., Khan, M. Z., & Ashraf, Z. (2025). LWLCM: A novel lightweight stream cipher using logistic chaos function and multiplexer for IoT communications. PLOS ONE, 20(9), Article e0330976. https://doi.org/10.1371/journal.pone.0330976

[29] İnce, C., İnce, K., & Hanbay, D. (2024). A multilayer nonlinear permutation framework and its demonstration in lightweight image encryption. Entropy, 26(10), Article 885. https://doi.org/10.3390/e26100885

[30] Sugio, N. (2024). Bit-based evaluation of lightweight block ciphers SLIM, LBC-IoT, and SLA by mixed integer linear programming. IET Information Security, 2024(1), Article 1741613. https://doi.org/10.1049/2024/1741613

[31] Wang, Y., Liang, Z., Fan, T., & Zhou, T. (2026). Improved differential cryptanalysis of the ultra-lightweight block cipher PICO. Entropy, 28(9), Article 1006. https://doi.org/10.3390/e28091006

[32] Ganesan, M. M., & Selvaraj, S. (2025). An improved key scheduling for advanced encryption standard with expanded round constants and non-linear property of cubic polynomials. International Journal of Electrical and Computer Engineering, 15(2), 2455–2467. https://doi.org/10.11591/ijece.v15i2.pp2455-2467

[33] Rukhin, A., et al. (2010). A statistical test suite for random and pseudorandom number generators for cryptographic applications (NIST Special Publication 800-22 Rev. 1a). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-22r1a