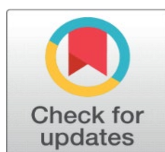
Original Article
ISSN (Online): 2454-1907

International Journal of Engineering Technologies and Management Research
April 2025 12(4), 13–24

ENHANCING AES-LIKE IOT SECURITY WITH DIVERSE S-BOX AND INVOLUTORY MATRIX IN MIXCOLUMNS TRANSFORMATIONS

Fu Jung Kan [1], Shui Hsiang Su [1], J-D Huang [2], T-K Zhvo [2], Yu- Ti Chang [2], Yan-Haw Chen [2]

[1] Department of Electronic Engineering, I-Shou University, Kaohsiung, Taiwan
[2] Department of Information Engineering, I-Shou University, Kaohsiung, Taiwan



ABSTRACT

Embedded systems are widely used in various fields, including device-to-device communication, vehicular and maritime mobility, and public infrastructure. These systems often involve the exchange and transmission of sensitive and critical information, which requires protection. However, these devices have limited hardware resources, necessitating compact size and low cost, which restricts the complexity of security algorithms. Therefore, an improved AES algorithm, specifically a lightweight AES-like method, is proposed. It enhances the SubBytes step through dynamic S-box lookups table and uses different 8×8 affine matrix transformations to scramble data. The proposed method is 31% faster than traditional approaches. Furthermore, in the Mix Columns transformation, the encryption process using 16×16 involutory matrix achieves 66% speed improvement over the matrix multiplication traditional approach. circulant matrix, while the branch number increases from 5 to 17. Finally, the encryption process also reduces decryption time.

Keywords: AES, Affine Matrix, Circulant Matrix, Involutory Matrixs

Received 08 February 2025
Accepted 04 March 2025
Published 17 April 2025

Corresponding Author

Yan-Haw Chen, yanchen@isu.edu.tw

DOI
10.29121/ijetmr.v12.i4.2025.1553

Funding: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Copyright: © 2025 The Author(s). This work is licensed under a Creative Commons Attribution 4.0 International License.

With the license CC-BY, authors retain the copyright, allowing anyone to download, reuse, re-print, modify, distribute, and/or copy their contribution. The work must be properly attributed to its author.

1. INTRODUCTION

In this paper, we improve the AES method to simplify both the encryption and decryption processes, achieving lightweight block encryption suitable for embedded systems. The first study on lightweight cryptography (LWC) methods was conducted in Eisenbarth et al. (2007). Devices are classified into two categories: ultra-lightweight and lightweight. Ultra-lightweight implementations are ideal for highly constrained devices (e.g., limited computation speed, memory size, and power consumption) that can execute traditional AES algorithms in software. Microprocessor-based devices, commonly used in daily life, have limited resources, which requires careful selection of data processing, communication protocols, and

underlying technologies to meet stringent operational requirements Fysarakis et al. (2015). Given that these devices often handle private or security-critical information, protecting this data from malicious attackers is essential, making secure cryptographic components vital. Research in lightweight cryptography (LWC) focuses on encryption algorithms tailored for constrained devices Hatzivasilis et al. (2016). Lightweight cryptography also uses elliptic curve cryptography Ning et al. (2024) for symmetric key exchange. Symmetric key algorithms are primarily used for encrypting large volumes of data, offering strong confidentiality, while asymmetric key algorithms are typically employed for message exchange between communicating parties, ensuring confidentiality, integrity checks, and authentication protocols. The widely known block cipher AES Donald et al. (2023) has become a standard encryption method, provided that the device meets the necessary resource constraints. AES is the standard symmetric key cipher used for encryption applications, and new block ciphers designed for this purpose are gaining popularity, introducing innovations and improving efficiency. In Manifavas et al. (2012), the authors evaluated software implementations of lightweight symmetric and asymmetric with hash functions cryptography. Roman (2007) explored lightweight hardware and software solutions for wireless sensor networks, a highly constrained hardware platform group. Paar et al. (2009) discussed new trends in lightweight hardware block ciphers and stream ciphers, while Kitsos et al. (2012) focused on the hardware architecture for implementation of block ciphers. Cazorla et al. (2013) conducted a fair comparison by implementing and evaluating lightweight block ciphers on the same platform. Dinu et al. (2015) introduced software implementations of lightweight block ciphers on three different platforms, and Anjali et al. (2012) carried out cryptanalysis attacks on lightweight block ciphers. This paper uses an 8×8 involutory matrix for encryption and decryption Wang et al. (2024), as it does not require the inverse matrix during decryption. The introduction of circulant matrices allows for fast computation of the inverse matrix, and we use an 8×8 circulant matrix for encryption Wang and Chen (2022),Wang et al. (2021). The proposed S-box method reduces the overall encryption execution time by approximately 30% compared to traditional AES encryption. By using a 16×16 involutory matrix, the breach number can increase to 17, significantly enhancing data confusion.

2. MATERIALS AND METHODS

The AES SubBytes steps of traditional method is as shown in Figure 1 The section will discuss how to speed inverse operation and diversity affine transformation in Figure 2

Figure 1

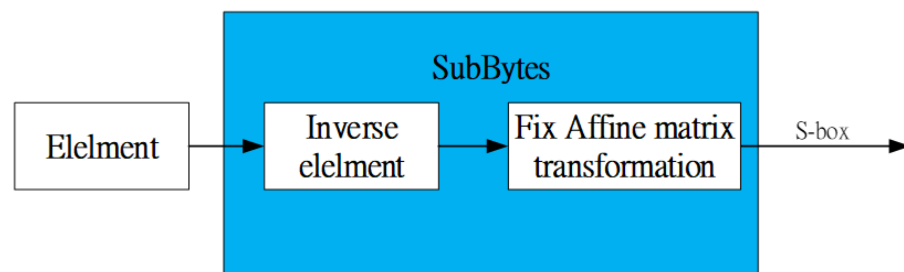


Figure 1 The Computing the Value of the S-Box

Figure 2

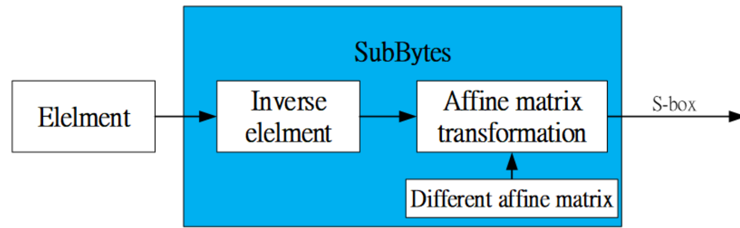


Figure 2 The Diversity Affine Matrix Computing the Value of the S-Box

2.1. FINITE FIELD OPERATION

Let the polynomial $A(x)=\sum_{i=0}^{m-1} a_i x^i$, $B(x)=\sum_{i=0}^{m-1} b_i x^i$, and the polynomial $C(x)=\sum_{i=0}^{m-1} c_i x^i$ be over $GF(2^m)$. The polynomial $f(x)$ is irreducible polynomial. The finite field operations are defined as follows:

Addition operation

$C(x)=A(x)\oplus B(x)$, where the operator represents the XOR operation.

Multiplication operation

$C(x)\equiv A(x)\cdot B(x) \bmod f(x)$.

```
def GFM (a, b):
   c = 0;  b1=[0,b]; f = [0, 27]
   for i in range(7, 0, -1):
    c = c ^ b1[(a>>i)&0x01]
   c = ((c << 1) & 0xff) ^ f[(c>>7)&0x01]
   c = c ^ b1[a&0x01]
   return c
```

Inverse operation

The inverse element of vector ***A*** in $GF(2^m)$ is derived using Fermat's Little Theorem, which is expressed as:

$$A^{-1}=A^{2^m-2}. \tag{1}$$

The inverse method based on Fermat's Little Theorem requires many finite field multiplications, specifically 2^m-2 multiplications, to calculate the inverse element. It needs more computation time for computing inverse. Therefore, we propose a new method for computing the inverse element over $GF(2^m)$, in Section 2.2.

Square operation

The square operation is simply in finite field because A raised to the power 2, the value of 2 is same as finite filed base p (i.e., $GF(p^m)$, p=2). Let A be the polynomial $A(x)=a_0+a_1x+\cdots+a_{m-1}x^{m-1}$, where $a_i\in GF(2)$. The square of the polynomial $A(x)$ is given by,

$$A^2 \equiv (a_0 + a_1 x^2 + \cdots + a_{m-1} x^{2(m-1)}) \text{m}$$ 2)

2.2. SPEED UP INVERSE OPERATOR

In (3), 2^m-2 can be used decomposition by number theory as follows:

$$2^m - 2 = 2^{m-1} + 2^{m-2} + \cdots + 2^1. \quad (3)$$

If m=8, then $2^8 - 2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 = 254$. The substitution $2^8 - 2$ into (1) is represented as,

$$A^{-1} = A^{2^7} \cdot A^{2^6} \cdot A^{2^5} \cdot A^{2^4} \cdot A^{2^3} \cdot A^{2^2} \cdot A^{2^1}. \quad (4)$$

The standard inversion is required 7 multiplications, the following equation (3) is presented a new method for computing inversion:

$$2^8 - 2 = \left(\left((2+1)2^2 + (2+1)\right)2^2 + (2+1)\right)2^2 + 2^1 = 254.$$

$$A^{-1} = \left((M^4 \cdot M)^4 \cdot M\right)^4 \cdot A^2, \quad (5)$$

where $M = A^2 \cdot A$. The proposed method in (5) can be represented as shown in Figure 3

Figure 3

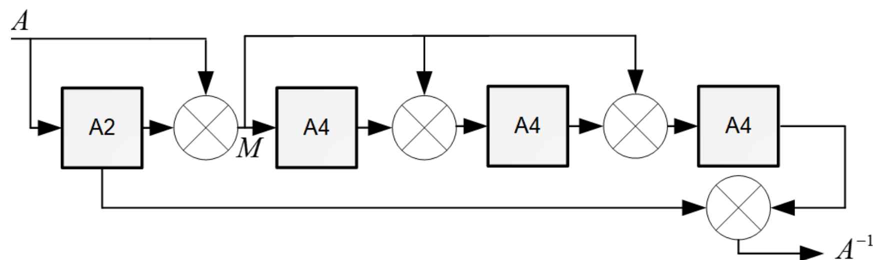


Figure 3 A New Method for Computing M=8 Element Inversion

The modifying inversion is only required 4 multiplications. Fig. 3 can be written a python program as below:

```
def inv(a):
    a2=A2(a)
    M=GFM (a2, a)
    IA=GFM(A4(GFM(A4(GFM(A4(M), M)), M)), a2)
    return IA
```

2.3. A2 AND A4 OPERATION

In (4), we can rewrite as follows:

$$A(x)^2 \equiv \sum_{i=0}^{m-1} a_i (x^2)^i \bmod f(x).$$

$$A(x)^4 \equiv \sum_{i=0}^{m-1} a_i (x^4)^i \bmod f(x).$$

Where x^2 is presented a vector (00000100) and x^4 is presented a vector (0001000) and $a_i \in GF(2)$. To construct the values of x^2 and x^4 for power i, see Table 1.

Table 1

Table 1 The Lookup Tables A^2 And A^4 is for Computing Inversion

Index i	$A(x)=\sum_{i=0}^{7} a_i x^i$	$A(x)^2$ TableA2[i]= x^{2i}	$A(x)^4$ TableA4[i]= x^{4i}
0	a_0	0x01	0x01
1	a_1	0x04	0x10
2	a_2	0x10	0x1b
3	a_3	0x40	0xab
4	a_4	0x1b	0x5e
5	a_5	0x6c	0x97
6	a_6	0xab	0xb3
7	a_7	0x9a	0xc5

For example, A=255=0xFF, the coefficients a_i of the polynomial $A(x)$ are 1 with Table 1 to addition all values A^2=0x13 and A^4=0x1a. The method is written by Python program as follows:

```
def A2(A):
    a2t=0
    for i in range (0,8):
      a2t=a2t^(((A>>i) &0x01) TableA2[i])
    return a2t
def A4(A):
    a4t=0
    for i in range (0,8):
       a4t=a4t^(((A>>i) &0x01) TableA4[i])
    return a4t
```

2.4. SPEED UP AFFINE MATRIX MULTIPLICATION

Consider an IoT system deployed in a smart home environment where various sensors and devices communicate that needs more security for devices. In AES SubBytes step is using 256 bytes memory size for S-box, this paper will use matrix divided into quarters for speeding diversity affine matrix in the SubBytes that can reduce to make lookup table time, and the performance of IoT devices are as below:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ \hline b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \left[\begin{array}{cccc|cccc} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\ a_7 & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \\ a_6 & a_7 & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ a_5 & a_6 & a_7 & a_0 & a_1 & a_2 & a_3 & a_4 \\ \hline a_4 & a_5 & a_6 & a_7 & a_0 & a_1 & a_2 & a_3 \\ a_3 & a_4 & a_5 & a_6 & a_7 & a_0 & a_1 & a_2 \\ a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_0 & a_1 \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_0 \end{array}\right] \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \hline b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}$$

According to multiplication matrix that can be $B_0' = A_0B_0 + A_1B_1$ and $B_1' = A_1B_0 + A_0B_1$. In the finite field, we can reduce multiplication matrix, elements addition property is the same matrix $A_0B_0 \oplus A_0B_0 = 2A_0B_0 = 0$. Therefore, we rewrite multiplication matrix into the equation (6).

$$\begin{bmatrix} B_0' \\ B_1' \end{bmatrix} = \begin{bmatrix} A_0B_0 + A_1B_1 + 2A_0B_1 \\ A_1B_0 + A_0B_1 + 2A_0B_0 \end{bmatrix} = \begin{bmatrix} A_0(B_0 + B_1) + (A_0 + A_1)B_1 \\ A_0(B_0 + B_1) + (A_0 + A_1)B_0 \end{bmatrix} = \begin{bmatrix} F + G \\ F + H \end{bmatrix}, \quad (6)$$

where $F = A_0\left(B_0 + B_1\right), G = \left(A_0 + A_1\right)B_1$ and $H = \left(A_0 + A_1\right)B_0$.

$$F = \begin{bmatrix} a_0 & a_7 & a_6 & a_5 \\ a_1 & a_0 & a_7 & a_6 \\ a_2 & a_1 & a_0 & a_7 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 + b_4 \\ b_1 + b_5 \\ b_2 + b_6 \\ b_3 + b_7 \end{bmatrix}, \quad (7)$$

$$G = \begin{bmatrix} a_0 + a_4 & a_7 + a_3 & a_6 + a_2 & a_5 + a_1 \\ a_1 + a_5 & a_0 + a_4 & a_7 + a_3 & a_6 + a_2 \\ a_2 + a_6 & a_1 + a_5 & a_0 + a_4 & a_7 + a_3 \\ a_3 + a_7 & a_2 + a_6 & a_1 + a_5 & a_0 + a_4 \end{bmatrix} \begin{bmatrix} b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix}, \quad 8)$$

and,

$$H = \begin{bmatrix} a_0 + a_4 & a_7 + a_3 & a_6 + a_2 & a_5 + a_1 \\ a_1 + a_5 & a_0 + a_4 & a_7 + a_3 & a_6 + a_2 \\ a_2 + a_6 & a_1 + a_5 & a_0 + a_4 & a_7 + a_3 \\ a_3 + a_7 & a_2 + a_6 & a_1 + a_5 & a_0 + a_4 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}. \quad (9)$$

Therefore, the computing F, G, and H matrix can be used smaller lookup tables, both G matrix and H matrix are the same operation. The using lookup tables for making S-box is as following python program:

SubBytes: Making S-box

```
sbox= [0] *256
def new_s_box(aff):
    CBIT= [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]
    A0= [0] 4; A= [0] 4; afft= [0] 4
    Aft [0] = aff;
    aft [1] = (aft [0]>>1) | ((aft [0] &0x01) <<7)
    aft [2] = (aft [1]>>1) | ((aft [1] &0x01) <<7)
    aft [3] = (aft [2]>>1) | ((aft [2] &0x01) <<7)

    A0[0] = aft [0]>>4; A0[1] =aft [1]>>4
    A0[2] = aft [2]>>4; A0[3] =aft [3]>>4
    A [0] = (aft [0] &0xf) ^A0 [0]; A [1] = (aft [1] &0xf) ^A0[1]
    A [2] = (aft [2] &0xf) ^A0[2]; A [3] = (aft [3] &0xf) ^A0[3]

    FT= [0] *16; GH= [0] *16
    for Bt in range (0,16):
        F=0; G=0
        for k in range (0,4):
            F=F|(CBIT[A0[k]&Bt] <<k)
            G=G|(CBIT[A[k]&Bt] <<k)
        FT[Bt]=F
        GH[Bt]=G

    for i in range (0,256):
        iv=newinv(i).
        B0=0; B1=0.
        for j in range (0,4): # reverse data
            B0=(B0<<1) |((iv>>j) &0x01).
            B1=(B1<<1) |((iv>>(j+4)) &0x01)
        Bt=B0^B1
        F=FT[Bt]
        G=GH[B1]
        H=GH[B0]
        sbox[i]=(((F^H)<<4)|(F^G))^0X63
```

In the Python program, the lookup table **FT[]** is computed using matrix multiplication (7), while the lookup table **GH[]** can be computed using both matrix multiplication (8) and matrix multiplication (9) for the S-box computation.

2.5. AFFINE MATRIX DETERMINANT

The circulant matrix, we must find inverse matrix to decrypt data. It has a **Theorem 1** as follows:

Theorem 1. $\det(Circ[a_0, a_1, a_2, \cdots, a_{2^n-1}]) = (a_0 + a_1 + a_2 + \cdots + a_{2^n-1})^{2^n}$, where n= 2, 3, 4,..., using n=3, if the determinant is not zero then it has an inverse of circulant matrices. The elements of the affine matrix are $a_i \in \{0,1\}$.

Using Theorem 1 can be find affined matrix that has inversion shown in Table 2

Table 2

Table 2 The Value of Aff is for Making Affine Matrix

No.	aff	No.2	aff3	No.4	aff5	No.6	aff7	No.8	aff9	No.10	aff11	No.12	aff13	No.14	aff15
0	1	16	20	32	40	48	61	64	80	80	a1	96	c1	112	e0
1	2	17	23	33	43	49	62	65	83	81	a2	97	c2	113	e3
2	4	18	25	34	45	50	64	66	85	82	a4	98	c4	114	e5
3	7	19	26	35	46	51	67	67	86	83	a7	99	c7	115	e6
4	8	20	29	36	49	52	68	68	89	84	a8	100	c8	116	e9
5	0b	21	2a	37	4a	53	6b	69	8a	85	ab	101	cb	117	ea
6	0d	22	2c	38	4c	54	6d	70	8c	86	ad	102	cd	118	ec
7	0e	23	2f	39	4f	55	6e	71	8f	87	ae	103	ce	119	ef
8	10	24	31	40	51	56	70	72	91	88	b0	104	d0	120	f1
9	13	25	32	41	52	57	73	73	92	89	b3	105	d3	121	f2
10	15	26	34	42	54	58	75	74	94	90	b5	106	d5	122	f4
11	16	27	37	43	57	59	76	75	97	91	b6	107	d6	123	f7
12	19	28	38	44	58	60	79	76	98	92	b9	108	d9	124	f8
13	1a	29	3b	45	5b	61	7a	77	9b	93	ba	109	da	125	fb
14	1c	30	3d	46	5d	62	7c	78	9d	94	bc	110	dc	126	fd
15	1f	31	3e	47	5e	63	7f	79	9e	95	bf	111	df	127	fe

If we use **aff = (8f)** $_{\textbf{hex}}$ **= (1,0,0,0,1,1,1,1)** $_{\textbf{b}}$, we obtain the same S-Box value in AES method.

2.6. MIXCOLUMNS STEPS

An involutory matrix is a square matrix that is its own inverseis for in AES MixColumns transformation with python program as follows:

$$A_{16\times16}=\left[\begin{array}{cccccccc|cccccccc}
01&03&04&05&06&07&08&09&0A&0B&0C&0D&0E&10&02&1E\\
03&01&05&04&07&06&09&08&0B&0A&0D&0C&10&0E&1E&02\\
04&05&01&03&08&09&06&07&0C&0D&0A&0B&02&1E&0E&10\\
05&04&03&01&09&08&07&06&0D&0C&0B&0A&1E&02&10&0E\\
06&07&08&09&01&03&04&05&0E&10&02&1E&0A&0B&0C&0D\\
07&06&09&08&03&01&05&04&10&0E&1E&02&0B&0A&0D&0C\\
08&09&06&07&04&05&01&03&02&1E&0E&10&0C&0D&0A&0B\\
09&08&07&06&05&04&03&01&1E&02&10&0E&0D&0C&0B&0A\\
\hline
0A&0B&0C&0D&0E&10&02&1E&01&03&04&05&06&07&08&09\\
0B&0A&0D&0C&10&0E&1E&02&03&01&05&04&07&06&09&08\\
0C&0D&0A&0B&02&1E&0E&10&04&05&01&03&08&07&06&07\\
0D&0C&0B&0A&1E&02&10&0E&05&04&03&01&09&08&07&06\\
0E&10&02&1E&0A&0B&0C&0D&06&07&08&09&01&03&04&05\\
10&0E&1E&02&0B&0A&0D&0C&07&06&09&08&03&01&05&04\\
02&1E&0E&10&0C&0D&0A&0B&08&09&06&07&04&05&01&03\\
1E&02&10&0E&0D&0C&0B&0A&09&08&07&06&05&04&03&01
\end{array}\right]$$

A0 A1
A1 A0

The product of two matrices A and B is denoted as $D = AB$, where $D=[d_0,d_1,d_2,d_3,d_4,d_5,d_6,d_7,d_8,d_9,d_{10},d_{11},d_{12},d_{13},d_{14},d_{15}]^T$ and

$$B=[b_0,b_1,b_2,b_3,b_4,b_5,b_6,b_7,b_8,b_9,b_{10},b_{11},b_{12},b_{13},b_{14},b_{15}]^T.$$

Using equation (6), matrix multiplication for matrix A can be simplified computing by M16() calls M8(), which in turn calls M4() for matrix multiplication.

M4(): for Computing 4×4 Matrix Multiplication

```
def M4(a,b):
d=[0] 4
t0=b[0]^b[2]; t1=b[1]^b[3]; t2=a[0]^a[1]; t3=GFM(a[0],t0^t1)
s0=a[0]^a[2]; s1=a[1]^a[3]; s2=GFM(s0,b[2]^b[3]); s3=s0^s1
r0=GFM(s0,b[0]^b[1]); r1=s0^s1
f0=t3^GFM(t2,t1); f1=t3^GFM(t2,t0)
g0=s2^GFM(s3,b[3]); g1=s2^GFM(s3,b[2])
h0=r0^GFM(r1,b[1]); h1=r0^GFM(r1,b[0])
d[0]=f0^g0; d[1]=f1^g1; d[2]=f0^h0; d[3]=f1^h1
return d
```

M8(): for Computing 8×8 Matrix Multiplication

```
def M8(a,b):
d=[0] 8; t=[0] 4; bt=[0] 4
t[0]=a[0]^a[4]; t[1]=a[1]^a[5]; t[2]=a[2]^a[6]; t[3]=a[3]^a[7]
bt[0]=b[0]^b[4]; bt[1]=b[1]^b[5]; bt[2]=b[2]^b[6]; bt[3]=b[3]^b[7]
F=M4(a[0:4],bt); G=M4(t,b[4:8]); H=M4(t,b[0:4])
d[0]=F[0]^G[0]; d[1]=F[1]^G[1]
d[2]=F[2]^G[2]; d[3]=F[3]^G[3]
d[4]=F[0]^H[0]; d[5]=F[1]^H[1]
d[6]=F[2]^H[2]; d[7]=F[3]^H[3]
return d
```

M16(): for Computing 16×16 Matrix Multiplication
def M16(a,b):
d=[0] 16; t=[0] 8; bt=[0] 8
t[0]=a[0]^a[8]; t[1]=a[1]^a[9]; t[2]=a[2]^a[10]; t[3]=a[3]^a[11]; t[4]=a[4]^a[12]
t[5]=a[5]^a[13]; t[6]=a[6]^a[14]; t[7]=a[7]^a[15]
bt[0]=b[0]^b[8]; bt[1]=b[1]^b[9]; bt[2]=b[2]^b[10]; bt[3]=b[3]^b[11]
bt[4]=b[4]^b[12]; bt[5]=b[5]^b[13]; bt[6]=b[6]^b[14]; bt[7]=b[7]^b[15]
F=M8(a[0:8],bt); G=M8(t,b[8:16]); H=M8(t,b[0:8])
d[0]=F[0]^G[0]; d[1]=F[1]^G[1]; d[2]=F[2]^G[2]; d[3]=F[3]^G[3]
d[4]=F[4]^G[4]; d[5]=F[5]^G[5]; d[6]=F[6]^G[6]; d[7]=F[7]^G[7]
d[8]=F[0]^H[0]; d[9]=F[1]^H[1]; d[10]=F[2]^H[2]; d[11]=F[3]^H[3]
d[12]=F[4]^H[4]; d[13]=F[5]^H[5]; d[14]=F[6]^H[6]; d[15]=F[7]^H[7]
return d

Therefore, the function M16() uses elements a0 through a15 in first rows of the matrix *A*, where their values correspond to 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0x10, 0x2, 0x1e, respectively. Therefore, we can call the function M16() for matrix multiplication as follows:

D=[0]*16; data=[0] 16 # List data corresponds to the values of vector B.

a=[1,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0x10,2,0x1E]

data=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] # Input data

D=M16(a,data) #D=[0x1c, 0x58, 0xfe, 0x8e, 0xcb, 0xac, 0xa9, 0xd2, 0x7f, 0xf0, 0x9d, 0x6, 0x43, 0x24, 0x21, 0x6a]

2.7. AES-LIKE ENCRYPTION

The modified AES, namely AES-like encryption using a different affine matrix, is shown in Figure 4 The matrix of diffusion data uses 16×16 involutory matrix in the AES-like MixColumns transformation which involves only one matrix multiplication.

Figure 4

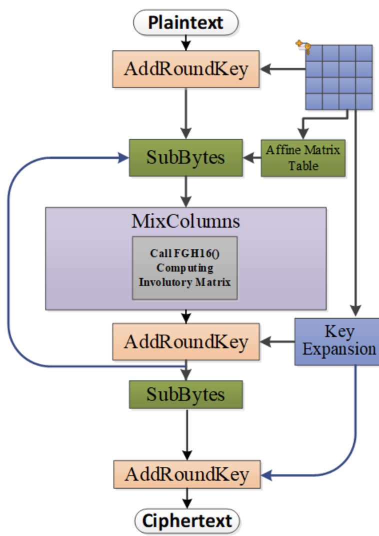


Figure 4 The 16×16 involutory Matric Using in the AES-Like

3. EXPERMENTAL RESULTS

The AES encryption procedure is modified by removing the ShiftRows transformation and incorporating a 16×16 involutory matrix. The resulting encryption and decryption process is referred to as AES-like, as shown in Fig. 4. A different affine matrix is used to generate the S-box lookup table, and its performance is evaluated in Table 3, demonstrating that the proposed method is 30% faster than the traditional approach. For AES-like encryption with a 128-bit key, running the cipher and invcipher 5000 times with the 16×16 matrix improves performance compared to the traditional 4×4 matrix multiplication, reducing computation time by approximately 30%, as shown in Table 4. Furthermore, the proposed matrix multiplication method achieves a 66% speed improvement over the traditional approach.

Table 3

Table 3 The Affine Matrix Multiplication to Make S-Box Performance.

Sub Bytes making S-Box method	Execution time
Traditional method	1.12s
The proposed method	0.79s

Table 4

Table 4 The Keys Lengths of 128bits for Encryption/Decryption Execution Time.

The encryption+decryption methods	Cipher+invCipher time
Using 4´4 involutory matrix for MixColumns steps by traditional method	2.16s
Using 16´16 involutory matrix for MixColumns steps by traditional method	9.10s
Using 16´16 involutory matrix for MixColumns steps by proposed method	3.12

4. CONCLUSIONS

This study shows that the processing complexity of matrix multiplication in GF(2^8) can be reduced by splitting the affine matrix into four submatrices and utilizing the cyclic convolution property. The proposed method achieves faster data processing compared to the traditional affine matrix transformation used for generating lookup tables. Additionally, using dynamic affine matrix calculations for dynamic S-boxes can enhance encryption security. When comparing encryption methods with a 128-bit key, the proposed method, implemented with a 16×16 involutory matrix, outperforms the AES 4×4 circulant matrix by approximately 30%. However, the number of breaches increases from 5 to 17. The AES-like algorithm removes the ShiftRows step from AES, as illustrated in Fig. 4, because the 16×16 matrix provides better diffusion performance than both traditional 4×4 matrix and 8×8 matrix. In the future, the affine matrix multiplication method can also be applied to hardware design to enhance efficiency by reducing the number of XOR logic gates.

CONFLICT OF INTERESTS

None.

ACKNOWLEDGMENTS

This study was supported in part by National Science and Technology Council NISC 113-2221-E-214-021.

REFERENCES

Anjali, A., Priyanka, & Pal, S. K. (2012). A Survey of Cryptanalytic Attacks on Lightweight Block Ciphers. International Journal of Computer, Science and Information & Security, 2.

Cazorla, M., Marquet, K., & Minier, M. (2013). Survey and Benchmark of Lightweight Block Ciphers for Wireless Sensor Networks. Iacr Cryptology Eprint Archive, 295.

Dinu, D., Corre, Y. L., Khovratovich, D., Perrin, L., Grobshadl, J., & Biryukov, A. (2015). Triathlon of Lightweight Block Ciphers for the Internet of Things. IACR Cryptology Eprint Archive, 209.

Donald L., Phillip J. Bond, Karen H. Brown,(2023) Standard, NIST FIPS. . Advanced Encryption Standard (AES). Federal Information Processing Standards Publication.

Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., & Uhsadel, L. (2007). A Survey of Lightweight-Cryptography Implementations. IEEE Design & Test of Computers, 24(6), 522-533.

Fysarakis, K., Hatzivasilis, G., Askoxylakis, I. G., & Manifavas, C. (2015). RT-SPDM: Realtime Security, Privacy & Dependability Management of Heterogeneous Systems. In Human Aspects of Information Security, Privacy and Trust (pp. 619-630). Springer.

Hatzivasilis, G., Floros, G., Papaefstathiou, I., & Manifavas, C. (2016). Lightweight Authenticated Encryption for Embedded on-Chip Systems. Information Security Journal, 25, 1-11.

Kitsos, P., Sklavos, N., Parousi, M., & Skodras, A. N. (2012). A Comparative Study of Hardware Architectures for Lightweight Block Ciphers. Computers & Electrical Engineering, 38 (1), 148-160.

Manifavas, C., Hatzivasilis, G., Fysarakis, K., & Rantos, K. (2012). Lightweight Cryptography for Embedded Systems: A Comparative Analysis. In 6th International Workshop on Autonomous and Spontaneous Security (pp. 333-349). Springer.

Ning, Y. D., Chen, Y. H., Shih, C. S., & Chu, S. I. (2024). Lookup Table-Based Design of Scalar Multiplication for Elliptic Curve. CRyptographycryptography, 8 (11), 1-16.

Paar, C., Poschmann, A., & Robshaw, M. J. B. (2009). New Designs in Lightweight Symmetric Encryption. RFID Security, 3, 349-371.

Roman, R., Alcaraz, C., & Lopez, J. A. (2007). Survey of Cryptographic Primitives and Implementations for Hardware-Constrained Sensor Network Nodes. Mobile Networks and Applications, 12 (4), 231-244.

Wang, J. J., & Chen, Y. H. (2022). The Inverse of Circulant Matrices Over GF(2m). Discrete Mathematics, 345 (3), 1-10.

Wang, J. J., Chen, Y. H., Chen, Y. W., & Lee, C. D. (2021). Diversity AES in MixColumns Step with 8×8 Circulant Matrix. International Journal of Engineering Technologies and Management Research, 8 (9), 19-35.