SOFTWARE IMPLEMENTATION OF SM4 BLOCK ENCRYPTION ALGORITHM IN PYTHON PROGRAMMING LANGUAGE

Liu Lingyun
Ph.D. student of the National University of Uzbekistan, Jining Normal University, Shenyu International Community, Jining District, Ulanqab, Inner Mongolia, China

Abstract

In this work, the methods of implementation of the SM4 encryption algorithm transformations in the Python programming language were studied. It was found that the implementation of S-box in the form of a 1-dimensional table allows to achieve efficiency in terms of time. An efficient method has also been found for the cyclic left shift operation. The results obtained for the code written in the Python programming language for the encryption algorithm were compared with other implementation options in terms of the time spent on encryption and decryption, and it was found that this implementation method allows to achieve higher efficiency in terms of time.

Keywords. SM4,SM4_Mix, symmetric-key cryptographic algorithm, round function, mixed substitution, nonlinear transformation, linear transformation, key generation, reference function

Introduction

The SM4 block cipher, a symmetric-key cryptographic algorithm, was introduced by the Office of State Commercial Cryptography Administration (OSCCA) in China and was officially recognized as the national standard for the cryptographic industry in March 2012 [1, 2]. In June 2021, it became part of the ISO/IEC 18033-3 international standard [3]. SM4 operates as a block cipher, featuring a block size of 128 bits and a key size of 128 bits, and it involves a total of 32 rounds. In each round, all four state words and one subkey word serve as inputs, replacing a single state word. The length of both state and subkey words is 32 bits.
SM4, the sole symmetric encryption algorithm authorized by OSCCA for Chinese use, has found extensive applications across various industries. Its main functions encompass bolstering network security by encrypting data packets and safeguarding communication protocols. Additionally, SM4 plays a pivotal role in electronic payment systems, guaranteeing the security and confidentiality of financial transactions. In the domain of image processing, SM4 ensures the privacy of visual

data through encryption, digital watermarking, and secure image sharing. Furthermore, the integration of SM4 into 5G communication systems enhances security in data transmission, network function virtualization (NFV), and edge computing.

The proper execution of cryptographic algorithms is a critical requirement. This paper is dedicated to the exploration of efficient implementations of SM4 on high-end platforms. The most obvious approach is to leverage instruction set extensions. An illustrative example is the incorporation of advanced encryption standard new instructions (AES-NI) [4] into numerous processors, leading to substantial enhancements in the speed and security of applications utilizing AES. However, there are limited processors that offer specialized instructions tailored for SM4, which significantly hinders the performance of applications that rely on it. Building on our prior findings in the paper authored by Miao et al. [5], we naturally propose further optimizations to accelerate the SM4 block cipher.

Main part

SM4 operations are defined with respect to 8-bit, 32-bit, or 128-bit quantities, allowing us to refer to 8-bit values as "bytes," 32-bit values as "words," and 128-bit values as "blocks." The symbol $\oplus$ signifies the bitwise exclusive-OR operation, while "<<<" denotes a left circular rotation by a certain number of bits in a 32-bit word vector. Block cipher encryption with the key k is represented as $\mathrm{Enc_k}$. When it comes to the multiplication of two elements $\mathrm{X, Y} \in \mathrm{GF}(2^{128})$, we use the notation $\mathrm{X \cdot Y}$, and the details of the field multiplication operation. The expression $\{0,1\}^{\mathrm{m}}$ signifies a bit string of length m, and 0^{128} represents a string consisting of 128 zero bits. The concatenation of two bit strings, A and B, is denoted as A||B. Several researches have been conducted on the implementation of the SM4 encryption algorithm [6-14].

SM4 is a block cipher algorithm with a 128-bit block size and a 128-bit key length. It employs an unbalanced Feistel structure and undergoes 32 rounds of the encryption process, with $\mathrm{X_i} \in \mathrm{Z_2^{32}}, \mathrm{i} = 0,1, \dots, 35$ representing bit strings of 32-bit length, respectively. Subsequently, SM4 employs a reverse transformation to produce the resulting ciphertext. The 32 round keys are sequentially generated using the key expansion algorithm based on the original 128-bit key. In the decryption phase, a comparable structure is employed, but the order of round keys is reversed [2].

Round Function

Assuming the input to the round function is represented as $(X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$ in a 32-bit space, and the round key is denoted as $rk \in Z_2^{32}$ in the same 32-bit space, the round function F can be stated as follows:

$$F(X_0, X_1, X_2, X_3) = (X_1, X_2, X_3, X_0 \oplus T(X_1, X_2, X_3 \oplus rk)). \qquad (1)$$

Mixed substitution T

$Z_2^{32} \rightarrow Z_2^{32}$ represents a reversible transformation that consists of both a nonlinear transformation τ and a linear transformation L. In simpler terms, it can be represented as $T(\cdot) = L(\tau(\cdot))$.

Transformation T, which is nonlinear in nature

τ is made up of four parallel S-boxes (S). Assuming $A = (a_0, a_1, a_2, a_3) \in (Z_2^8)^4$ represents the input to τ and $B = (b_0, b_1, b_2, b_3) \in (Z_2^8)^4$ represents the resulting output, then the relationship can be expressed as follows:

$$B = (b_0, b_1, b_2, b_3) = \tau(A) = (S(a_0), S(a_1), S(a_2), S(a_3)). \qquad (2)$$

Linear Transformation L

The 32-bit result generated by the nonlinear transformation τ serves as the input for the linear transformation L. Assuming $B \in Z_2^{32}$ represents the input to L and $C \in Z_2^{32}$ is the resulting output, then the relationship can be described as follows:

$$C = L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24). \qquad (3)$$

Methodology

In the Python programming language, substitution tables (S-box) can be represented in three ways. These are in the form of dictionary view (Sbox_dict), 1 (Sbox_Table1) and 2 (Sbox_Table1) dimensional arrays. According to these methods of expression, the methods of referring to these tables also differ.

According to the form of representation of the substitution table, the functions to refer to them are as follows:

1. A reference function for a state expressed in a dictionary view:

```
def tau_dict(input: int) -> int:
    a = dict()
    b = dict()
    a[0] = input // 0x01000000
    a[1] = (input & 0x00ff0000) >> 16
    a[2] = (input & 0x0000ff00) >> 8
    a[3] = input % 0x100
    b[0] = Sbox_dict[a[0]]
    b[1] = Sbox_dict[a[1]]
    b[2] = Sbox_dict[a[2]]
    b[3] = Sbox_dict[a[3]]
    return (b[0] << 24) + (b[1] << 16) + (b[2] << 8) + b[3]
```

2. A reference function for a state expressed as a 1-dimensional array:

```
def tau1(input: int) -> int:
    a0 = input // 0x01000000
    a1 = (input & 0x00ff0000) >> 16
    a2 = (input & 0x0000ff00) >> 8
    a3 = input % 0x100
    b0 = Sbox_Table1[a0]
    b1 = Sbox_Table1[a1]
    b2 = Sbox_Table1[a2]
    b3 = Sbox_Table1[a3]
     return (b0 << 24) + (b1 << 16) + (b2 << 8) + b3
```

3. A reference function for a state expressed as a 2-dimensional array:

```
def tau2(a):
    b1 = Sbox_Table2[(a & 0xf0000000) >> 28][(a & 0x0f000000) >> 24]
    b2 = Sbox_Table2[(a & 0x00f00000) >> 20][(a & 0x000f0000) >> 16]
            b3 = Sbox_Table2[(a & 0x0000f000) >> 12][(a & 0x00000f00) >>  8]
            b4 = Sbox_Table2[(a & 0x000000f0) >>  4][(a & 0x0000000f) >>  0]
    return (b1 << 24) | (b2 << 16) | (b3 << 8) | (b4 << 0)
```

Using the following code, results were obtained to compare the time spent on these expression methods and reference functions:

```
l2=2**32
t1=time.time()
for i in range(10000000):
        d=random.randint(0,l2)
        s_dict=tau_dict(d)
t2=time.time()
print(t2-t1)
t1=time.time()
for i in range(10000000):
        d=random.randint(0,l2)
        s1=tau1(d)
```

```
t2=time.time()
print(t2-t1)
t1=time.time()
for i in range(10000000):
        d=random.randint(0,l2)
        s2=tau2(d)
t2=time.time()
print(t2-t1)
```

The values of the time taken to refer the substitution table for 10,000,000 random 32-bit values using the above program code are shown in Table 1 below.

Table 1 The values of the time taken to refer the substitution table for 10,000,000 random 32-bit values

№	The representation method of S_box	A reference function	Time spent (sec)
1.	dictionary view (Sbox_dict)	def tau_dict(input: int) -> int	16.1635637283325
2.	1 dimensional arrays (Sbox_Table1)	def tau1(input: int) -> int	12.3912270069122
3.	2 dimensional arrays (Sbox_Table2)	def tau2(a)	14.3787536621093

The data in Table 1 indicates that employing a 1-dimensional array to represent substitution tables in the Python programming language results in significantly less time consumption compared to using a dictionary or a 2-dimensional array for representation.

The cyclic left shift operation can also be expressed in different ways in the Python programming language. These methods are mentioned below.

1. 1st method operation cyclic shift left

```
def left_shift_1(a: int, n: int) -> int:
    for i in range(n):
        a <<= 1
        if a // 0x100000000 == 1:
            a %= 0x100000000
            a += 1
    return a
```

2. 2nd method operation cyclic shift left

```
def left_shift_2(a:int, n:int):
    size=32
```

```
   n =n% 32
   return (a << n) | (a >> (size - n)) &0xffffffff
```

3. 3rd method operation cyclic shift left

```
def left_shift_2(a:int, n:int):
   size=32
   n =n% 32
   return (a << n) ^ (a >> (size - n)) &0xffffffff
```

4. MixColumns()

```
        import time
        def mixColumns(input):
       a = input // 0x01000000
       b = (input & 0x00ff0000) >> 16
       c = (input & 0x0000ff00) >> 8
       d = input % 0x100
      a1=mul2[a] ^ mul3[b] ^ c ^ d
      b1=a ^ mul2[b] ^ mul3[c] ^ d
      c1=a ^ b ^ mul2[c] ^ mul3[d]
      d1=mul3[a] ^ b ^ c ^ mul2[d]
     #print()
```

5. MixColumns() with Lut_table

```
        def gmul(a, b):
        if b == 1:
          return a
       tmp = (a << 1) & 0xff
         if b == 2:
      return tmp if a < 128 else tmp ^ 0x1b
        if b == 3:
      return gmul(a, 2) ^ a
        t1=time.time()
      for i in range(100000000):
   mixColumns(0xA3B1BAC6)
     t2=time.time()
     print(t2-t1)
```

Table 2 shows the time taken by the above three options to perform a left-cycling swipe operation on 100,000,000 random integers.

Table 2 Time taken to performs a cyclic shift left and Mixcolumns() transformation of 100,000,000 random 32-bit integers

No.	The representation method of S_box	Time spent (sec)
1.	1st method	403.48764538764954
2.	2nd method	142.70483493804932
3.	3rd method	142.435124874115
4.	MixColumns()	687.4968481063843
5.	MixColumns() with Lut_table	134.1191828250885

The line MixColumns() with Lut_table describes the time taken to display the results of MixColumns using a pre-calculated table.

Table 2 that it is appropriate to use the 3rd method in the Python programming language to perform the cyclic shift operation to the left. The most time was spent on method 1. A little less time was spent in method 3 compared to method 2 due to the or and xor operations. In Table 3.2 above, you can see that the MixColumns() rendering is faster than the cyclic scroll left rendering when using precomputed tables.

Below are the main functions of the SM4 encryption algorithm written in the Python programming language.

Python code for L transformation:

```
def L (B: int) -> int:
    return B ^ left_shift_3(B, 2) ^ left_shift_3(B, 10) ^ left_shift_3(B, 18)
^left_shift_3(B, 24)
```

Python code for T transformation:

```
def T(a: int) -> int:
   return L(tau1(a))
```

Python code for L′ transformation:

```
def L_key(B: int) -> int:
   return B ^ left_shift_3(B, 13) ^ left_shift_3(B, 23)
```

Python code for T′ transformation:

```
def T_key(a: int) -> int:
   return L_key(tau1(a))
```

Python code of the key generation algorithm:

```
def key_generation(MK: tuple) -> list:
   K = dict()
   K[0] = MK[0] ^ FK[0]
```

```
    K[1] = MK[1] ^ FK[1]
    K[2] = MK[2] ^ FK[2]
    K[3] = MK[3] ^ FK[3]
    for i in range(32):
        K[i+4] = K[i] ^ T_key(K[i+1] ^ K[i+2] ^ K[i+3] ^ CK[i])
    rk = list(K.values())
    return list(rk[4:])
```

Python code for F transformation:

```
def F(X: list, rk: int) -> list:
    return X[0] ^ T(X[1] ^ X[2] ^ X[3] ^ rk)
```

Code of the encryption process in Python:

```
def encrypt_function(X: tuple, rk: tuple) -> tuple:
    X_list = list(X)
    for i in range(32):
        X_list.append(F(X_list[i:i+4], rk[i]))
    Y = (X_list[35], X_list[34], X_list[33], X_list[32])
    return Y
```

Code of the decryption process in Python:

```
def decrypt_function(X: tuple, rk: tuple) -> tuple:
    X_list = list(X)
    for i in range(32):
        X_list.append(F(X_list[i:i+4], rk[31-i]))
    Y = (X_list[35], X_list[34], X_list[33], X_list[32])
    return Y
```

The time taken to encrypt and decrypt 100,000 blocks of plaintext using this software code was compared with other implementations. Table 3.3 shows the comparison results.

Table 3.3 The time it takes to encrypt and decrypt 100,000 blocks

No.	Implementation	Time spent (sec) (SM4)	
		Encryption process	Decryption process
1.	[77]	36.23312	36.06769
2.	[85]	9.26895	10.14109
3.	[84]	6.723578	6.760374
4.	Our method (SM4)	5.526601	5.759675
5.	SM4_Mix	18.53468	18.74289
5.	SM4_Mix (Lut_table)	5.234876	5.482364

From the results in Table3.3, we can see that the implementation method proposed in this work has a time-efficient performance compared to other proposed implementation methods.

Conclusion

In this work, the methods of implementation of the SM4 encryption algorithm transformations in the Python programming language were studied. It was found that the implementation of S-box in the form of a 1-dimensional table allows to achieve efficiency in terms of time. An efficient method has also been found for the cyclic left shift operation. The results obtained for the code written in the Python programming language for the encryption algorithm were compared with other implementation options in terms of the time spent on encryption and decryption, and it was found that this implementation method allows to achieve higher efficiency in terms of time.

References

1. GM/T 0002-2012: SM4 Block Cipher Algorithm, “State cryptography administration of the People’s Republic of China,” March 2012.
   View at: Google Scholar
2. R. H. Tse, W. K. Wong, and M.-J. O. Saarinen, “The SM4 blockcipher algorithm and its modes of operations,” April 2018, Internet Engineering Task Force (IETF) https://datatracker.ietf.org/doc/html/draft-ribose-cfrg-sm4-10.
   View at: Google Scholar
3. ISO/IEC 18033-3: 2010/AMD1:2021, “Information technology-security techniques-encryption algorithms-part3: block ciphers-amendment1: SM4,” June 2021, https://www.iso.org/standard/81564.html.
   View at: Google Scholar
4. S. Gueron, “Intel advanced encryption standard (AES) new instructions set,” May 2010, Intel White Paper, Rev, 3:1-81.
   View at: Google Scholar
5. X. Miao, C. Guo, M. Wang, and W. Wang, “How fast can SM4 be in software,” in Information Security and Cryptology, Yi Deng and M. Yung, Eds., vol. 13837 of Lecture Notes in Computer Science, pp. 3–22, Springer Nature Switzerland, Cham, 2023.
   View at: Publisher Site | Google Scholar

6. Software.Intel.Com, "Haswell new instruction descriptions," http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/.
View at: Google Scholar
7. Intel Corporation, "Intel C++ compiler classic developer guide and reference," https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.
View at: Google Scholar
8. L. Wang, Z. Gong, Z. Liu, J. Chen, and H. Fu, "Fast software implementation of SM4 based on tower field," Journal of Cryptologic Research, vol. 9, no. 6, pp. 1081–1098, 2022.
View at: Google Scholar
9. X. Zhang, H. Guo, X. Zhang, C. Wang, and J. Liu, "Fast software implementation of SM4," Journal of Cryptologic Research, vol. 7, no. 6, pp. 799–811, 2020.
View at: Google Scholar
10. H. Lang, L. Zhang, and W. Wu, "Fast software implementation of SM4," Journal of University of Chinese Academy of Sciences, vol. 35, no. 2, pp. 180–187, 2018.
View at: Google Scholar
11. J. Zhang, M. Ma, and P. Wang, "Fast implementation for SM4 cipher algorithm based on bit-slice technology," in Smart Computing and Communication. SmartCom 2018, M. Qiu, Ed., vol. 11344 of Lecture Notes in Computer Science, pp. 104–113, Springer, Cham, 2018.
View at: Publisher Site | Google Scholar
12. http://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=7803DE42D3BC5E80B0C3E5D8E873D56A)
13. https://github.com/CCWUCMCTS/SM4/blob/main/SM4.py
14. https://github.com/windard/sm4/blob/master/Python/sm4.py