Hindawi
IET Information Security
Volume 2023, Article ID 1821499, 10 pages
https://doi.org/10.1049/2023/1821499

Research Article

Bit-Sliced Implementation of SM4 and New Performance Records

Xin Miao,[1] **Lu Li**,[1,2] **Chun Guo**,[1,3,4] **Meiqin Wang**,[1,2,4] **and Weijia Wang**[1,2,4]

[1]*School of Cyber Science and Technology, Shandong University, Qingdao 266237, Shandong, China*
[2]*Quan Cheng Laboratory, Jinan 250103, Shandong, China*
[3]*Shandong Research Institute of Industrial Technology, Jinan, Shandong 250102, China*
[4]*Key Laboratory of Cryptologic Technology and Information Security of Ministry of Education, Shandong University, Qingdao 266237, Shandong, China*

Correspondence should be addressed to Weijia Wang; wjwang@sdu.edu.cn

Received 31 May 2023; Revised 2 August 2023; Accepted 25 August 2023; Published 27 October 2023

Academic Editor: Tom Chen

SM4 is a popular block cipher issued by the Office of State Commercial Cryptography Administration (OSCCA) of China. In this paper, we use the bit-slicing technique that has been shown as a powerful strategy to achieve very fast software implementations of SM4. We investigate optimizations on two frontiers. First, we present a more efficient bit-sliced representation for SM4, which enables running 64 blocks in parallel with 256-bit registers. Second, we describe an optimized algorithm for data form transformations, also allowing efficient implementations of SM4 under Counter (CTR) mode and Galois/Counter mode. The above optimizations contribute to a significant performance gain on one core compared with the state-of-the-art results. This work is an extension of the conference paper at Inscrypt 2022, awarded the best paper award.

1. Introduction

SM4 block cipher is a symmetric-key cryptographic algorithm issued by the Office of State Commercial Cryptography Administration (OSCCA) of China and was identified as the national cryptographic industry-standard in March 2012 [1, 2]. It was incorporated into the ISO/IEC 18033-3 international standard in June 2021 [3]. SM4 is a block cipher with a 128-bit block size and a 128-bit key size, and it consists of 32 rounds. Each round uses all four state words and one subkey word as inputs, replacing a single state word. The length of one state/subkey word is 32-bit.

As the only OSCCA-approved symmetric encryption algorithm for use in China, SM4 has been applied to many industries. Its primary uses include network security for encrypting data packets and securing communication protocols. It also plays a vital role in electronic payment systems, ensuring the security and confidentiality of financial transactions. In image processing, SM4 ensures the privacy of visual data through encryption, digital watermarking, and secure image sharing. Moreover, SM4's integration into 5G communication systems enhances security in data transformation, network function virtualization (NFV), and edge computing.

An appropriate implementation is a very important requirement for cryptographic algorithms. In this paper, we focus on investigating the fast implementation of SM4 on high-end platforms. The natural thought is to use the instruction set extension. A typical example is the advanced encryption standard new instructions (AES-NI) [4], which has been integrated into many processors and has significantly improved the speed and security of applications with AES. However, few processors integrate instructions specially for SM4, which largely restricts the speed of applications collocating with it. Based our previous results in paper by Miao et al. [5], we naturally raise further optimizations to speed up SM4 block cipher.

1.1. Contributions. In this paper, we describe an improved bit-sliced implementation of SM4 based on an enhanced single instruction multiple data (SIMD) instruction set advanced vector extensions 2 (AVX2), which is an expansion of the AVX instruction set introduced in Intel's Haswell microarchitecture

Table 1: Comparison results of software implementations on Intel platforms.

Platform	Throughput (Gbps)	Method
Intel Xeon E5-2620 @2.40 GHz [11]	0.054	Bit-slicing
Intel Core i7-5500U @2.40 GHz [10]	1.75	Look-up table
Intel Core i7-6700 @3.40 GHz [10]	2.38	Look-up table
Intel Core i7-7700HQ @2.80 GHz [9]	2.52	Bit-slicing
Intel ore i7-11800H @2.30 GHz [8]	6.52	Bit-slicing
Ours (Intel Core i7-8700 @3.20 GHz)	7.63	Bit-slicing

[6, 7] (a.k.a., Haswell new instructions). Our improved implementation of SM4 runs at a speed of ≈2.40 cycles per byte (cpb) and ≈15.26 Gbits per second (Gbps) on one core with disabled hyper-threading and enabled turbo boost. Even if we take data form transformations into consideration, the bit-sliced implementation of SM4 can reach at the speed of ≈7.63 Gbps for throughput and ≈3.09 cpb for timing. To the best of our knowledge, they are both new speed records and outperform state-of-the-art software implementations, and also they operates in constant time. Indeed, the bit-sliced SM4 can be further improved using AVX-512 with ultrawide 512-bit vector operations capabilities to back up higher performance computing in theory. We still choose to consider AVX2 now since it is much more widely deployed (than AVX-512). To get a remarkable performance gain, we investigate optimizations on the following frontiers.

(i) First of all, we propose a new bit-sliced representation (that is the way to pack internal states of multiple blocks within the YMM registers) allowing to process 64-SM4 blocks efficiently with 256-bit registers.

(ii) Besides, according to the above bit-sliced representation, we describe an optimized algorithm for data form transformations and try to carry out transforming work at minimal cost.

Finally, we also provide an optimized algorithm for the data form transformation, and then give complete implementations of SM4 under Counter (CTR) mode and Galois/Counter mode (GCM). This work is an extension of the conference paper by Miao et al. [5] (awarded the best paper award). We specifically highlight below the novel aspects and extensions incorporated into this manuscript:

(1) We present an improved bit-sliced representation of the SM4 algorithm, which facilitates the optimization of data form transformations. The optimized data form transformation is a general algorithm capable of efficiently transforming any 128-bit data block and its inverse with low-performance overhead. In previous work by Miao et al. [5], to achieve favorable performance outcomes, distinct methods were employed for forward and backward transformations. Our optimized data form transformation unifies the approach for both forward and backward transformations, thereby obviating the need to differentiate between them. As a consequence, the process of converting data between the block-wise form and the bit-slicing-compatible form is significantly simplified.

(2) We significantly reduce the overhead of the data form transformation, mainly by adopting the method given in https://github.com/kste/skinny_avx for the bit-sliced implementation of SKINNY block cipher.

(3) We provide new performance records for SM4-CTR, SM4-CTR$^+$, SM4-GCM, and SM4-GCM$^+$. The efficiency of SM4-CTR and SM4-GCM has shown a substantial improvement of over 60% and 46%, respectively, compared to the results presented in the conference paper. This notable enhancement strongly indicates the performance advantage of the optimized data form transformation. These findings carry important implications and underscore the significance of the optimized data form transformation for achieving superior efficiency in the bit-sliced implementations.

(4) We compare performances of the bit-sliced implementations, among other block ciphers.

1.2. Related Works. The fast software implementations of SM4 have been investigated for several years, due to the wide applications such as networking software and operating system modules. Wang et al. [8] proposed an efficient SIMD-oriented optimization for *S*-box of SM4, and their fast software implementation reached 6.52 Gbps with AVX512 and single thread on an Intel 2.30 GHz processor. Zhang et al. [9] presented a fast software implementation of SM4 by exploiting bit-slicing technique with AVX2, where 256 blocks are processed in parallel. Their bit-sliced SM4 code ran at the throughput of 2,580 Mbps on an Intel 2.80 GHz processor. Lang et al. [10] presented an enhanced software implementation of SM4 with the performance of 1,795 Mbps and 2,437 Mbps on different Intel processors, respectively. Zhang et al. [11] proposed a bit-sliced software implementation of SM4 and detailed how to implement efficient transformation from original storage form to bit-sliced storage form on a 64-bit machine and carry out parallel encryption of multiple blocks. A brief comparison between our bit-sliced implementation and state-of-the-art works on Intel platforms is shown in Table 1 where we have also marked our cost of transformations, and more details will be given in Section 5. Last but not least, compared with the known bit-sliced implementations [8, 9, 11], ours (in addition to the significantly faster speed) is the first design to run n blocks in parallel by using $4n$-bit registers.

1.3. Pros and Cons. In this section, we will discuss the pros and cons of our improved bit-sliced implementation of SM4. The primary strength lies in its remarkable ability to efficiently process multiple (e.g., 64) SM4 blocks in parallel, making it a highly suitable choice for applications requiring the encryption of substantial data volumes using SM4. Conversely, the bit-sliced approach exhibits certain constraints when applied to short message encryption due to its inherent nature. Encryption tasks involving smaller data sizes may not be as effectively addressed by this method. However, the true potential of its parallel processing benefits shines through in scenarios where larger data sets necessitate encryption. Nevertheless, we believe that, in the case of encrypting a relatively large amount of bits, a fast implementation of encryption is usually significant for the performance of the application as well. Therefore, the improved bit-sliced SM4 is an excellent choice for applications, such as 5G, image sharing, and wireless networks, which require encrypting substantial amounts of data.

1.4. Organizations. Below we first present backgrounds in Section 2. We then present our strategy of bit-slicing SM4 in Section 3. In Section 4, we introduce the optimized data form transformation and describe the implementations of modes. The results and comparisons are shown in Section 5. Finally, Section 6 concludes the whole paper.

2. Backgrounds

2.1. Notations. In the following, we agree on the conventions used throughout the rest of this paper, mainly focussing on the block cipher encryption and its modes of operation. All operations of SM4 are defined over 8-bit, 32-bit, or 128-bit quantities so that 8-bit values can simply be called bytes, 32-bit values words and 128-bit values blocks. The symbol $\oplus$ denotes the bitwise exclusive-or operation and $\lll$ means a left circular rotation by bits in a 32-bit word vector which is different from its specific definitions in Section 3. The block cipher encryption with the key k is denoted as Enc_k. The multiplication of two elements $X, Y \in GF(2^{128})$ is denoted as $X \cdot Y$, and the field multiplication operation is defined in Section 2.4. The expression $\{0, 1\}^m$ denotes the bits string with length m and 0^{128} represents a string of 128 zero bits. The concatenation of two bit strings A and B is represented as $A \parallel B$.

2.2. The SM4 Block Cipher. SM4 is a block cipher algorithm whose block size and key length are both 128 bits. It adopts an unbalanced Feistel structure and iterates its round function 32 times during the encryption phase, where $X_i \in Z_2^{32}$, $i = 0, 1, \ldots, 35$ represents a bit string of length 32 bits, respectively. Finally, SM4 applies the reverse transformation to produce the corresponding output ciphertext. The 32 round keys are generated in turn by the key expansion algorithm with the original 128-bit key. The decryption phase has a similar structure except that the order of round keys needs to be reversed [2].

2.2.1. Round Function. Suppose the input to the round function is $(X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$, and the round key is $rk \in Z_2^{32}$, then the round function F can be expressed as follows:

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

2.2.2. Mixed Substitution T. $Z_2^{32} \longrightarrow Z_2^{32}$ is an invertible transformation, composed of a nonlinear transformation τ and a linear transformation L. That is, $T(\cdot) = L(\tau(\cdot))$.

2.2.3. Nonlinear Transformation τ. τ is composed of 4 S-boxes in parallel. Suppose $A = (a_0, a_1, a_2, a_3) \in (Z_2^8)^4$ is the input to τ and $B = (b_0, b_1, b_2, b_3) \in (Z_2^8)^4$ is the corresponding output, then

$$\begin{aligned} B &= (b_0, b_1, b_2, b_3) = \tau(A) \\ &= (S\text{-box}(a_0), S\text{-box}(a_1), S\text{-box}(a_2), S\text{-box}(a_3)). \end{aligned} \tag{2}$$

2.2.4. Linear Transformation L. The 32-bit output from the nonlinear transformation τ is the input to the linear transformation L. Suppose the input to L is $B \in Z_2^{32}$, and the corresponding output is $C \in Z_2^{32}$, then

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

2.3. The CTR Mode. The CTR mode is a confidentiality mode of operation that features the application of the forward cipher to a set of input blocks, called counter blocks, to produce a sequence of output blocks that are exclusive OR (XOR) with the plaintext to produce the ciphertext, and vice versa [12]. The "nonce" portion and the "counter" portion should be concatenated together to constitute counter blocks (e.g., storing the nonce in the upper 96 bits and the counter in the lower 32 bits of a 128-bit counter block). The sequence of counter block values must be different from every other one of them. This condition is not restricted to a single message, but all of the counter blocks should be distinct. Given a range of counter blocks $T_0, T_1, \ldots, T_{n-1}$ and plaintext $P_0, P_1, \ldots, P_{n-1}$, CTR encryption leaving out padding can be defined as follows:

(1) Forward cipher $O_j = Enc_k(T_j)$, for $j = 0, 1, \ldots, n-1$.
(2) Ciphertext $C_j = P_j \oplus O_j$, for $j = 0, 1, \ldots, n-1$.

In CTR encryption, the forward cipher functions can be performed in parallel. Moreover, the forward cipher functions can be applied to the counter block values prior to the availability of the plaintext data.

2.4. The GCM. GCM is one of the most widely used authenticated encryption schemes designed by McGrew and Viega [13]. It is constructed from a block cipher with a block size of 128 bits, such as the AES algorithm. It combines the CTR mode with a block cipher-based Wegman–Carter message authentication code (MAC) in an encrypt-then-MAC manner. The MAC employs a universal hash function defined over a binary Galois field [14]. However, GCM does not follow generic composition, and the establishment of its provable security is the outcome of an intricate line of works [13, 15, 16].

We focus on the (authenticated) encryption function of GCM. In addition, we mainly focus on the GCM variant with (fixed length) 96-bit nonces. It is mandated (e.g., RFC 4106 or IPsec [17], RFC 5647 or SSH [18], and RFC 5288 or SSL [18]) or recommended (e.g., RFC 5084 [19] and 5116 [20]) in many standards to use fixed-length nonces with 96 bits. In this respect, the encryption function GCM $.\mathrm{Enc}_k(N, M)$ takes a nonce $N \in \{0,1\}^{96}$ and a message $M \in \{0,1\}^*$ as the inputs. It first encrypts M to C with CTR mode $\mathrm{GCTR}_k(N, M)$, where the initial counter block value is the concatenation of N and the integer 2. Then, it invokes hash function $\mathrm{GHASH}_H(C)$ to have the digest of C, where $H := \mathrm{Enc}E_k(0^{128})$ is the secret hash key generated by encrypting the zero block.

Write C as multiple 128-bit blocks $C = (C_1, C_2, \ldots, C_n)$. Then,

$$\begin{aligned}\mathrm{GHASH}_H(C) &= \sum_{j=1}^{n} C_j H^{n-j+1}\\ &= C_1 \cdot H^n \oplus C_2 \cdot H^{n-1} \oplus \ldots \oplus C_n \cdot H,\end{aligned} \tag{4}$$

where $\cdot$ stands for multiplications over the field $GF(2^{128})$ constructed by the irreducible polynomial $P = x^{128} + x^7 + x^2 + x + 1$ [21]. Also in [21], appropriate methods are provided for us to directly invoke and calculate GHASH_H of GCM.

Alternatively, $\mathrm{GHASH}_H(C)$ can be computed by repeating

$$Y_i = [(C_i \oplus Y_{i-1}) \cdot H] \bmod P\,(P = x^{128} + x^7 + x^2 + x + 1), \tag{5}$$

for $i = 1, \ldots, n$, where $Y_i, i = 1, \ldots, n$ are outputs of the function GHASH_H, and modular is taken over the aforementioned field $GF(2^{128})$. Eventually, the authentication tag T with the length of t bits is derived by truncating $\mathrm{Enc}_k(N \parallel 1) \oplus \mathrm{GHASH}_H(C)$ to t bits.

2.5. A Nomenclature for AVX2 and More. Haswell is Intel's microarchitecture based on the 22-nm process for mobile, desktops, and servers. It introduced a number of new instructions, such as AVX2, BMI1, BMI2, and MOVBE. AVX2 (Advanced Vector Extensions 2), also known as Haswell new instructions, is an expansion of the AVX instruction set. Its prominent features include the introduction of 256-bit wide YMM registers, enabling SIMD operations on 256-bit data elements. This enhancement allows for parallel processing of multiple data elements within a single instruction, leading to improved computational performance. AVX2 primarily focuses on integer vectorization, offering a set of SIMD instructions for integer arithmetic, bitwise operations, shuffles, and blends. Furthermore, AVX2 finds extensive application in diverse fields, including image processing, encryption/decryption, signal processing, among others, where highly parallel data processing is essential. AVX-512 bit are 512-bit extensions to the 256-bit AVX SIMD instructions for x86 instruction set architecture (ISA) proposed by Intel in July 2013. This wider vectorization capability further boosts data parallelism, allowing simultaneous processing of even more data elements in a single instruction.

YMM registers are 256-bit wide vector registers introduced with AVX2, enabling simultaneous processing of multiple data elements within a single instruction. On the other hand, ZMM registers are introduced with AVX-512. With double the width of YMM registers, ZMM registers further extend the data parallelism capabilities. Both YMM registers and ZMM registers are essential components of the SIMD architecture, facilitating efficient vectorized processing and significantly accelerating computations involving large datasets. We still choose to consider AVX2 in this work since it is much more widely deployed than AVX-512.

3. Bit-Slicing SM4

The concept of bit-sliced implementation is to convert the algorithm into a series of logical bit operations (e.g., XOR and AND gates) and process multiple encryption blocks in parallel. In order to enhance the software performance of SM4 when implemented in a bit-sliced style on 256-bit platforms, we consider a modified bit-sliced representation for SM4 on the basis of bit-sliced scheme in paper by Miao et al. [5], and then describe changes in the applications to multiple bit-sliced blocks.

3.1. A Modified Bit-Sliced Representation of SM4. By Miao et al. [5], the bit-sliced representation of SM4 is introduced, based on the unbalanced Feistel structure of SM4 and considering both general-purpose registers and memory accesses. The construction of bit-sliced representation involves separating the bits of the same row and packing the bits of the same column. Moreover, multiple blocks are arranged at regular intervals rather than directly in tandem, as depicted in Figure 1. The bit-sliced representation is extended to encompass 64 blocks, with the data organized in 256-bit registers, as illustrated in Figure 2. The 256-bit registers are denoted as slice n $(n = 0, 1, \ldots)$, and $b^0, b^1, \ldots, b^{63}$ represent the 64 blocks, where b_i^j denotes the i-th bit of the j-th block.

Our modified bit-sliced representation of SM4 here continues the line of design in paper by Miao et al. [5]. In order to facilitate the optimized data form transformation algorithm described later in Section 4, we would like to group all 64 blocks into even eight parts, where we term every part as "bundle" (32-bit) and each bundle is comprised of eight blocks, more specifics as illustrated in Figure 3.

Slice 0 (256-bit) stores the least significant bit of column 0 from all 64 input blocks, and it will get loaded into one register when necessary. Labels of the different blocks in the bit-sliced representation or this kind of grouping are related to the arrangement of those initial blocks, but there would be no effect on the encryption process.

We are able to use this representation because most AVX2 instructions are strict with the operations crossing lanes freely but can manipulate quadword (64-bit) or doubleword (32-bit) values as individual processing units. The

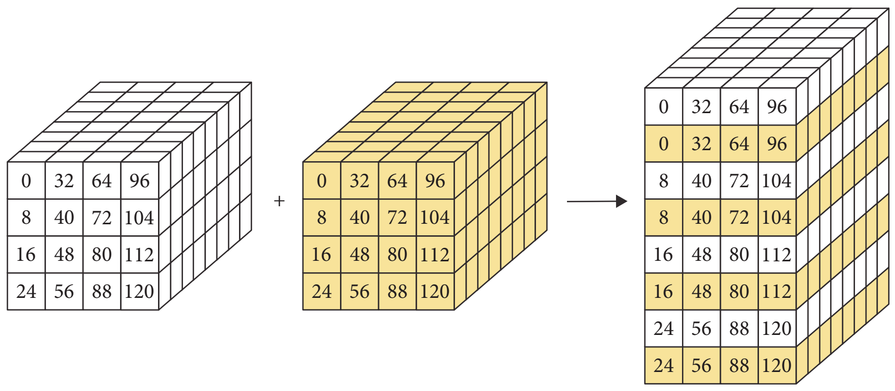

FIGURE 1: The bit-sliced representation of two SM4 blocks.

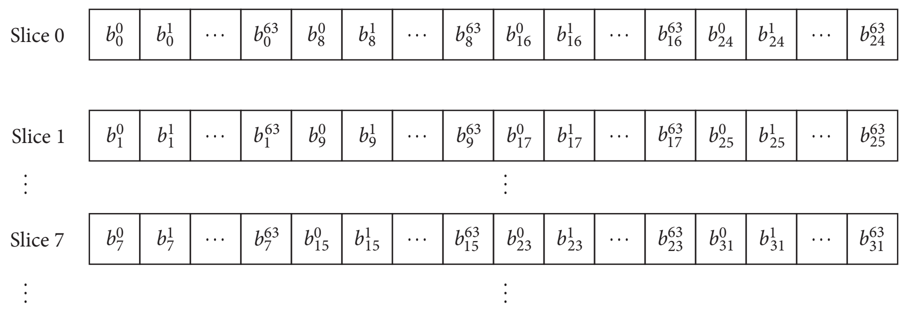
Slice 0
b_0^0 b_0^1 ⋯ b_0^{63} b_8^0 b_8^1 ⋯ b_8^{63} b_{16}^0 b_{16}^1 ⋯ b_{16}^{63} b_{24}^0 b_{24}^1 ⋯ b_{24}^{63}
Slice 1
b_1^0 b_1^1 ⋯ b_1^{63} b_9^0 b_9^1 ⋯ b_9^{63} b_{17}^0 b_{17}^1 ⋯ b_{17}^{63} b_{25}^0 b_{25}^1 ⋯ b_{25}^{63}
Slice 7
b_7^0 b_7^1 ⋯ b_7^{63} b_{15}^0 b_{15}^1 ⋯ b_{15}^{63} b_{23}^0 b_{23}^1 ⋯ b_{23}^{63} b_{31}^0 b_{31}^1 ⋯ b_{31}^{63}

FIGURE 2: The bit-sliced representation of 64 SM4 blocks in 256-bit registers.

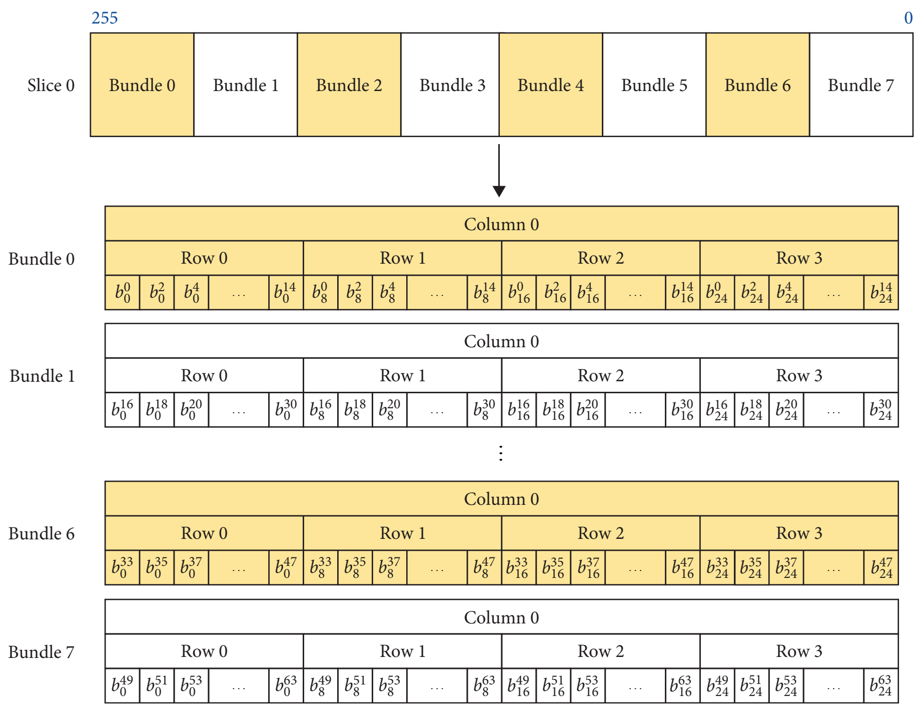
255
0
Slice 0
Bundle 0
Bundle 1
Bundle 2
Bundle 3
Bundle 4
Bundle 5
Bundle 6
Bundle 7
Bundle 0
Column 0
Row 0
Row 1
Row 2
Row 3
b_0^0 b_0^2 b_0^4 … b_0^{14} b_8^0 b_8^2 b_8^4 … b_8^{14} b_{16}^0 b_{16}^2 b_{16}^4 … b_{16}^{14} b_{24}^0 b_{24}^2 b_{24}^4 … b_{24}^{14}
Bundle 1
Column 0
Row 0
Row 1
Row 2
Row 3
b_0^{16} b_0^{18} b_0^{20} … b_0^{30} b_8^{16} b_8^{18} b_8^{20} … b_8^{30} b_{16}^{16} b_{16}^{18} b_{16}^{20} … b_{16}^{30} b_{24}^{16} b_{24}^{18} b_{24}^{20} … b_{24}^{30}
Bundle 6
Column 0
Row 0
Row 1
Row 2
Row 3
b_0^{33} b_0^{35} b_0^{37} … b_0^{47} b_8^{33} b_8^{35} b_8^{37} … b_8^{47} b_{16}^{33} b_{16}^{35} b_{16}^{37} … b_{16}^{47} b_{24}^{33} b_{24}^{35} b_{24}^{37} … b_{24}^{47}
Bundle 7
Column 0
Row 0
Row 1
Row 2
Row 3
b_0^{49} b_0^{51} b_0^{53} … b_0^{63} b_8^{49} b_8^{51} b_8^{53} … b_8^{63} b_{16}^{49} b_{16}^{51} b_{16}^{53} … b_{16}^{63} b_{24}^{49} b_{24}^{51} b_{24}^{53} … b_{24}^{63}

FIGURE 3: The bit-sliced representation of 64 SM4 blocks for 256-bit platforms, where we take the 256-bit slice 0 to process 64 blocks $b^0, b^1, \ldots, b^{63}$ in parallel for example and b_i^j refers to the i-th bit of the j-th block.

$$R'_7 = R_7 \oplus R_5 \oplus (R_5 \lll 8) \oplus (R_5 \lll 16) \oplus (R_7 \lll 24)$$
$$R'_6 = R_6 \oplus R_4 \oplus (R_4 \lll 8) \oplus (R_4 \lll 16) \oplus (R_6 \lll 24)$$
$$R'_5 = R_5 \oplus R_3 \oplus (R_3 \lll 8) \oplus (R_3 \lll 16) \oplus (R_5 \lll 24)$$
$$R'_4 = R_4 \oplus R_2 \oplus (R_2 \lll 8) \oplus (R_2 \lll 16) \oplus (R_4 \lll 24)$$
$$R'_3 = R_3 \oplus R_1 \oplus (R_1 \lll 8) \oplus (R_1 \lll 16) \oplus (R_3 \lll 24)$$
$$R'_2 = R_2 \oplus R_0 \oplus (R_0 \lll 8) \oplus (R_0 \lll 16) \oplus (R_2 \lll 24)$$
$$R'_1 = R_1 \oplus (R_7 \lll 8) \oplus (R_7 \lll 16) \oplus (R_7 \lll 24) \oplus (R_1 \lll 24)$$
$$R'_0 = R_0 \oplus (R_6 \lll 8) \oplus (R_6 \lll 16) \oplus (R_6 \lll 24) \oplus (R_0 \lll 24)$$

Figure 4: The calculation of L for 64 SM4 blocks, where $R_i \lll j$ refers to a circular rotation of j bits to the left within a 32-bit bundle instead of within a 256-bit slice, updated to R_i', and i represents the i-th bit of each byte.

arrangement of one bundle is similar to the barrel shifter design [22] enabling efficient circular rotations with SHUFFLE instruction instead of SHIFT instruction.

3.2. The Applications to Multiple SM4 Blocks. Then, according to the above modified bit-sliced representation, we explain the application of the round function with the precomputed (and stored in the memory) round keys. The round encryption function F comprises XOR operations and mixed substitution T made up of a nonlinear layer τ (4 S-boxes in parallel) and a linear transformation L. Nevertheless, only the linear transformation L in a bit-sliced style after S-box is different from any of previous left circular rotations. Calculations of the other operations could totally follow the directions and details in paper by Miao et al. [5], and the implementation of S-box is no exception. On the 256-bit platform which can process 64 blocks or eight bundles in parallel, the operations of L are described in Figure 4.

4. Implementations of SM4-CTR, SM4-GCM, and More

The bit-slicing technique can benefit modes that support the parallel implementation of block ciphers such as CTR mode and GCM, moreover, this bit-sliced implementation needs a nonstandard data form. Hence, additional transformations of the data between the block-wise form and the bit-slicing-compatible form are required and considered to be expensive [23], which we call the *optimized data form transformation* in the rest of this paper, to distinguish it from the *data form transformation* in paper [5]. We term the block-wise data into the bit-sliced representations as *forward transformation* and its inverse as *backward transformation*. In addition, considering that the overheads of even optimized data form transformations are nonnegligible, we have been in use the variants of CTR and GCM that do not require the backward transformation, named CTR$^+$ mode and CTR$^+$ mode, respectively.

Considering the significant performance overhead of the data form transformation, we have opted to use the shortcut data form transformation for the forward transformation in [5]. However, the shortcut data form transformation is not a general algorithm to transform the data between the block-wise form and the bit-slicing-compatible form. It utilizes the pattern of the fixed nonce and incremented counters in the forward transformation, for example, the common setting under CTR mode is that the 96-bit "nonce" portions of 128-bit counter block values are fixed and the 32-bit "counter" portions are incremented (block-by-block) from 0 to 1. However, the input of the backward transformation does not have the pattern that the shortcut data form transformation required. Therefore, the backward transformation adopted the data form transformation by Miao et al. [5]. It is one of the main motivations that we propose the optimized data form transformation to unify the data transformation approach used in both forward and backward transformations. Moreover, the optimized data form transformation has the advantage of low-performance overhead.

4.1. The Optimized Data Form Transformation Algorithm. In this respect, we learn from the data form transformation algorithm for SKINNY [24] to present the optimized data form transformation algorithm for SM4. Before introducing the optimized data form transformation algorithm, we first present the initial data form of block-wise (with 128-bit blocks) data consisting of 4×4 bytes in total and how they are stored on 256-bit platforms as shown in Figure 5. For any 128-bit block, array of bytes can be represented as S [0]|| S [1]|| S[2]||…|| S [13] and they are loaded column-wise rather than in the row-wise. Then, each 256-bit register are large enough to cover two contiguous 128-bit blocks. Also, based on this arrangement, the first half of a 256-bit register is occupied by the first 128-bit block, and the second half is occupied by the second consecutive 128-bit block. As a result, for all 64 blocks $b^0, b^1, \ldots, b^{63}$, blocks whose labels are even numbers ($b^0, b^2, \ldots, b^{60}, b^{62}$) will be put into the former four 32-bit bundles, and the latter four 32-bit bundles are naturally used to hold odd numbers ($b^1, b^3, \ldots, b^{61}, b^{63}$).

Specifically, the optimized data form transformations include the forward transformation and the backward transformation. Taking the forward transformation algorithm that transforms any block-wise (with 128-bit blocks) data into the bit-sliced representations as detailed in Section 3 for example, we give a concrete description about this transformation. The entire transformation process is achieved mainly by the SWAPMOVE routine which is defined below and responsible for swapping the bits masked by M in B with the bits masked by M $\ll$ Num in A. According to the definition in the Listing, the early bit permutations can be simply implemented by means of bit swaps between the bytes, where we can constantly readjust the values of mask ("M" in the Listing) to locate different bits and use SHIFT instruction ("$\ll$") for efficient circular rotations. When setting the values of mask different from each individual doubleword value in order to realize the bit swaps between the bundles, we change to use SHUFFLE instruction to cross individual lanes (32-bit or 64-bit) and compute rotations.

Listing 1.1. C code for the SWAPMOVE routine.
```
Temp = (B ^ (A << Num)) & M;
B = B ^ Temp;
A = A ^ (Temp >> Num);
```

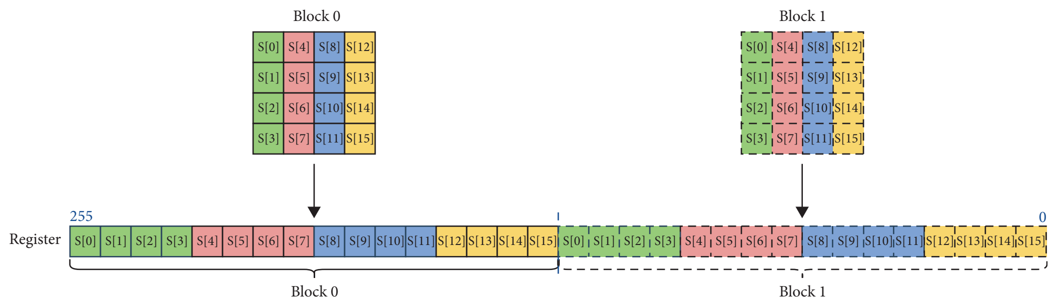

FIGURE 5: The initial 128-bit blocks and the storage behavior.

TABLE 2: Our results of different modes for software implementations.

	Mode	Timing (cpb)	Data form transformation
This work	SM4	3.09	Forward trans.: `optimized DFT` Backward trans.: `optimized DFT`
	SM4$^+$	2.40	Forward trans.: `optimized DFT` Backward trans.: none
	SM4-CTR	3.24	Forward trans.: `optimized DFT` Backward trans.: `optimized DFT`
	SM4-CTR$^+$	2.90	Forward trans.: `optimized DFT` Backward trans.: none
	SM4-GCM	5.54	Forward trans.: `optimized DFT` Backward trans.: `optimized DFT`
	SM4-GCM$^+$	5.22	Forward trans.: `optimized DFT` Backward trans.: none
[5]	SM4-CTR	8.14	Forward trans.: `shortcut DFT` Backward trans.: `DFT`
	SM4-CTR$^+$	2.70	Forward trans.: `shortcut DFT` Backward trans.: none
	SM4-GCM	10.35	Forward trans.: `shortcut DFT` Backward trans.: `DFT`
	SM4-GCM$^+$	5.10	Forward trans.: `shortcut DFT` Backward trans.: none

Noting that, the backward transformation is important as well. For example, in CTR mode, the bit-sliced representations that have been encrypted should transform back into the initial data form of 128-bit blocks and then perform XOR operations with the plaintext to generate the final ciphertext. Therefore, the optimized data form transformation algorithm need to be called twice. Obviously the backward transformation is the reverse of the forward one. That is to say, in fact, the only difference between the two directions of transformation is just to reverse the order of multiple SWAPMOVE routines (different from the values of mask).

4.2. Bit-slicing-Friendly Variants of CTR Mode and GCM. As for the parellel modes of operation, the 128-bit counter block values are first transformed (by the forward transformation) to fully comply with the bit-sliced encryption and then transformed back (by the backward transformation). GCM with the bit-sliced encryption is similar. With a view to the optimized data form transformation can not be omitted in computation cost, we propose a variant of CTR mode that flushes out the data form transformation at the outputs of parallel block ciphers, namely the backward transformation. The outputs of the parallel block ciphers, whether the bit-sliced representation or the block-wise representation, are uniformly distributed. Therefore, the security of the variant should be the same as the original CTR mode. We adopt the same strategy to GCM, resulting in a variant called GCM$^+$, for which we elaborate more formal details and its security proof in the appendix from paper by Miao et al. [5]. In the case of the implementation of GHASH step in GCM, namely the polynomial operations of GCM, we mainly refer to the code samples given by Gueron and Kounavis [21] which have already detailed the computation of Galois Hash.

5. Implementation Results and Comparisons

When considering an architecture for implementing bit-sliced SM4, Intel architecture emerges as a favorable choice. Intel's optimized instruction sets, particularly AVX2 and AVX-512, are specifically tailored for SIMD operations, making them highly suitable for the parallel-intensive nature of SM4 bit-slicing. This architectural alignment, coupled with Intel's widespread market support and mature development toolchain, facilitates efficient code optimization and maintainability. Consequently, opting for Intel architecture for bit-sliced SM4 implementations is a well-grounded decision, promising enhanced performance, and broad applicability. Given the broader deployment of AVX2 compared to AVX-512, we have decided to prioritize AVX2 for our consideration in implementing bit-sliced SM4. In this section, we will present the results of our improved bit-sliced SM4 implementation and compare them with some reference results.

5.1. The Software Performances of SM4. Evaluations are conducted separately for different modes as shown in Table 2. The criterion of these performances is clock cpb, i.e., the number of cycles required per byte of ciphertext, as this is the common metric used in the literature for software implementations [25]. Our bit-sliced SM4 enabled running in constant time and processing 64 blocks (1 KB) in parallel reaches 2.40 cpb for timing without considering any data form transformations. Additionally, our software performances are obtained when we disable the hyper-threading but enable the turbo boost. Moreover, `DFT` and `shortcut DFT` is an abbreviation for *the data form transformation* and *the shortcut data form transformation*, respectively. They are the two algorithms adopted by Miao et al. [5] to transform the data between the block-wise form and bit-sliced-compatible form. `Optimized DFT` represents *the optimized data form transformation* proposed in this work.

We can see that the performances of SM4$^+$, SM4-CTR$^+$, and SM4-GCM$^+$ without full data form transformations are significantly faster than those of SM4, SM4-CTR, and SM4-GCM with both forward and backward transformation in this work. Moreover, according to the conclusions by Beierle et al. [26], the input counter block values have the pattern of the fixed nonce and incremented counters, which can be known in advance and provided in correct bit-sliced representations to save the costs for data form transformation. As a consequence, the performances of CTR and GCM could be better in practice.

Upon comparing the performance of this work by Miao et al. [5], the efficiency of SM4-CTR and SM4-GCM has significantly improved by more than 60% and 46%, respectively. Moreover, the performance of SM4-CTR$^+$ and SM4-GCM$^+$ in this work is comparable to that by Miao et al. [5]. In pursuit of achieving better efficiency, Miao et al. [5] employed a custom data transformation method, i.e., the shortcut data form transformation, leveraging the specific pattern of input data in the forward transformation. In contrast, the optimized data transformation proposed in this work is a more general approach, which can be applied to any block-wise data (with 128-bit block) into the bit-sliced representation without requiring it to have a specific format. Therefore, considering both the performance and generalization of the optimized data transformation algorithm, this work is interesting. This advancement not only enhances the overall efficiency of SM4 but also contributes to a more cohesive and streamlined approach to data transformation in the realm of bit-sliced representations.

Table 3: Comparisons of bit-sliced implementations for some familiar block ciphers.

Block cipher	Timing (cpb)	Parallelization (block)
SM4	3.09	64
SKINNY-128-128 [26]	3.43	64
SIMON-128-128 [27]	2.21	64
AES-128-128 [4]	1.28	64

5.2. The Comparison of Block Ciphers. In this section, we try to find out how the block ciphers can be implemented in software and create a relatively level field for our results and other remarkable results of the different block ciphers. More precisely, we mainly consider the latest Intel processors using SIMD instruction sets to perform efficient parallel computations of input blocks and refer to the performance figures for bit-sliced implementations of these block ciphers in particular [26].

In Table 3, we present the performances of SKINNY [26], SIMON [27], and AES [4] in cycles per byte, alongside SM4. The bit-sliced implementation of AES exhibits the fastest efficiency. It is worth noting that AES's performance is optimized by utilizing the AES-NI instructions, which were specifically designed by Intel and AMD and integrated into the x86 instruction set using dedicated hardware circuits. Consequently, comparing the performance of other block ciphers with AES may not be entirely equitable due to the influence of the AES-NI instruction set. In the subsequent discussion, our primary focus will be on the comparison between SKINNY, SIMON, and SM4.

They all with preexpanded subkeys prior to encrypting blocks and targeting the same instruction set AVX2, except SIMON-128-128, the bit-sliced implementations take both the costs of the data form transformations and the costs of the actual encryptions into account. Hence, in comparison, SIMON does not include the cost of packing or unpacking the initial data, which prevents the meaningful comparison with the other two block ciphers having the same level of security. Compared to SKINNY, our bit-sliced implementation of SM4 demonstrates a performance advantage. Nevertheless, this rough comparison comes to conclusions that it still makes sense to optimize block ciphers for the specific platforms, rather, we can not deny that tackling the root of the problem is simplifying the algorithm to perform better just like SIMON with a compact enough design. These desired results declare that efficient bit-sliced implementations are possible and worthwhile in many situations, but it seems wise to be acutely aware of their drawbacks such as relatively expensive overheads of data form transformations [28].

6. Conclusions and Future Works

In this paper, we push the software implementation of SM4 to its limits with AVX2 instructions by investigating optimizations on multiple frontiers. First, we present a modified bit-sliced representation for SM4 that enables running 64 blocks in parallel with 256-bit registers efficiently. Second, we introduce an optimized data form transformation which can sharply reduce its overhead in bit-sliced implementations. Thanks to those optimizations, we can report our new bit-sliced SM4 to reach at the speed of ≈3.09 cpb for timing (with precomputed round keys and data form transformations), becoming the performance record of SM4 ever made on the Intel platforms. These significant improvements also demonstrate that the bit-slicing technique is actually promising on platforms with the enhanced SIMD architecture from practical points of view.

We also employ the optimized data form transformation algorithm for complete and efficient bit-sliced implementations of SM4, keeping full compatibility with existing parellel modes of operation, for example, the CTR mode and GCM. Furthermore, the expensive overhead on transforming data between the bit-slicing-compatible form and the block-wise form motivates us to adjust CTR mode and GCM to the bit-sliced implementation, resulting in bit-slicing-friendly variants of these two modes with an essential security proof.

While our work only concentrates on the platform with AVX2 instructions, we believe our optimizations for SM4 could bring about improvements on other architectures as well. Specifically, our method is applicable to various platforms, provided that the target architecture's register length is a multiple of 4 bits. This adaptability ensures the feasibility of employing our approach for bit-sliced SM4 implementation, offering potential advancements in performance and efficiency across a broader spectrum of the computing environments. Also, the number of general-purpose registers is limited (16 general-purpose registers available on our target platform), and thus numerous memory accesses dominate the entire SM4 processing. In this respect, we deem optimizing the number of memory accesses or register arrangement for the bit-sliced implementation as a valuable future study. We are also fired up about the implementation of SM4 on different platforms such as ARMv8/v9 for wider applicability of these techniques, and we will incorporate this throughout our following works. Another interesting topic might be the power analysis of our implementation, i.e., investigating the impact of the bit-sliced structure to the known attacking methods such as the chosen plaintext differential power analysis [29].

Data Availability

The source code used to support the findings of this study are available from the corresponding author upon request.

Disclosure

This work is an extension of the conference paper [5].

Conflicts of Interest

The authors have no conflicts of interest with regard to this work.

Authors' Contributions

Xin Miao and Lu Li contributed equally to this work. Xin Miao, Lu Li, Chun Guo, and Weijia Wang contributed in the investigation. Xin Miao and Lu Li contributed in the software. Weijia Wang, Xin Miao, and Lu Li contributed in the writing and methodology. Lu Li, Meiqin Wang, and Weijia Wang contributed in the validation. Chun Guo, Meiqin Wang, and Weijia Wang contributed in the resources and funding. Weijia Wang contributed in the conceptualization.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (Nos. 2021YFA1000600 and 2018YFA0704702), the National Natural Science Foundation of China (Grant Nos. 62002202, 62002204, and 62032014), the Program of Qilu Young Scholars (Grant Nos. 61580089963177 and 61580082063088) of Shandong University, the Program of Taishan Young Scholars of the Shandong Province, the Major Basic Research Project of Natural Science Foundation of Shandong Province, China (Grant No. ZR202010220025) and Quan Cheng Laboratory (Grant No. QCLZD202306).

References

[1] GM/T 0002-2012: SM4 Block Cipher Algorithm, "State cryptography administration of the People's Republic of China," March 2012.

[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.

[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.

[4] S. Gueron, "Intel advanced encryption standard (AES) new instructions set," Intel White Paper, Rev, 3:1-81, May 2010.

[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.

[6] Software.Intel.Com, "Haswell new instruction descriptions," http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/.

[7] Intel Corporation, "Intel C++ compiler classic developer guide and reference," https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[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.

[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.

[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.

[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.

[12] M. Dworkin, "Recommendation for block cipher modes of operation: methods and techniques," National Institute of Standards and Technology, December 2001.

[13] D. A. McGrew and J. Viega, "The security and performance of the Galois/counter mode (GCM) of operation," in *Progress in Cryptology-INDOCRYPT 2004*, A. Canteaut and K. Viswanathan, Eds., vol. 3348 of *Lecture Notes in Computer Science*, pp. 343–355, Springer, Berlin, Heidelberg, 2004.

[14] M. Dworkin, "Recommendation for block cipher modes of operation: galois/counter mode (GCM) and GMAC," National Institute of Standards and Technology, November 2007.

[15] T. Iwata, K. Ohashi, and K. Minematsu, "Breaking and repairing GCM security proofs," in *Advances in Cryptology–CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, pp. 31–49, Springer, Heidelberg, 2012.

[16] Y. Niwa, K. Ohashi, K. Minematsu, and T. Iwata, "GCM security bounds reconsidered," in *Fast Software Encryption. FSE 2015*, G. Leander, Ed., vol. 9054 of *Lecture Notes in Computer Science*, pp. 385–407, Springer, Heidelberg, 2015.

[17] J. Viega and D. McGrew, "The use of galois/counter mode (GCM) in ipsec encapsulating security payload (ESP)," Technical report, RFC 4106, June, 2005.

[18] K. Igoe and J. Solinas, "AES galois counter mode for the secure shell transport layer protocol," IETF Request for Comments, 5647, 2009.

[19] R. Housley, "Using AES-CCM and AES-GCM authenticated encryption in the cryptographic message syntax (CMS)," Technical report, RFC 5084, November, 2007.

[20] D. McGrew, "An interface and algorithms for authenticated encryption," Technical report, RFC 5116, January, 2008.

[21] S. Gueron and M. E. Kounavis, "Intel carry-less multiplication instruction and its usage for computing the GCM Mode," Intel Corporation, May 2010.

[22] A. Adomnicai and T. Peyrin, "Fixslicing AES-like ciphers: new bitsliced AES speed records on arm-cortex M and RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 1, pp. 402–425, 2021.

[23] M. Matsui and J. Nakajima, "On the power of bitslice implementation on intel core2 processor," in *Cryptographic Hardware and Embedded Systems-CHES 2007*, P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, pp. 121–134, Springer, Berlin, Heidelberg, 2007.

[24] S. Kölbl and T. Peyrin, "SKINNY family of block ciphers," November 2017, https://sites.google.com/site/skinnycipher/.

[25] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK families of lightweight block ciphers," Cryptology ePrint Archive, 2013, page 404, https://eprint.iacr.org/2013/404.

[26] C. Beierle, J. Jean, S. Kölbl et al., "The SKINNY family of block ciphers and its low-latency variant MANTIS," in *Advances in Cryptology-CRYPTO. 2016*, M. Robshaw and J. Katz, Eds., vol. 9815 of *Lecture Notes in Computer Science*, pp. 123–153, Springer, Berlin, Heidelberg, 2016.

[27] L. Wingers, "Software for SUPERCOP benchmarking of SIMON and SPECK," 2015, https://github.com/lrwinge/simon_speck_supercop.

[28] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, San Francisco, CA, USA, 2015.

[29] S. Wang, D. Gu, J. Liu, Z. Guo, W. Wang, and S. Bao, "A power analysis on SMS4 using the chosen plaintext method," in *2013 Ninth International Conference on Computational Intelligence and Security*, pp. 748–752, IEEE, Emeishan, China, 2013.