Received December 9, 2021, accepted December 27, 2021, date of publication January 4, 2022, date of current version January 13, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3140242

Speeding Up LAT: Generating a Linear Approximation Table Using a Bitsliced Implementation

GIYOON KIM[1], YONGJIN JEON[1], AND JONGSUNG KIM[1,2]
[1]Department of Mathematics and Financial Information Security, Kookmin University, Seoul 02707, South Korea
[2]Department of Information Security, Cryptology and Mathematics, Kookmin University, Seoul 02707, South Korea

Corresponding author: Jongsung Kim (jskim@kookmin.ac.kr)

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-00520, Development of SCR-Friendly Symmetric Key Cryptosystem and Its Application Modes).

ABSTRACT The substitution box (S-box) is one of the major components of cryptographic algorithms. An important issue for cryptographic algorithm designers in ensuring sufficient security from linear cryptanalysis, one of the most powerful attacks, is finding an S-box with a sufficiently low linear spectrum. However, to the best of our knowledge, most of the published S-box analysis tools cannot generate linear approximation tables for large S-boxes, such as 16-bit S-boxes. Even tools that support the generation of 16-bit linear approximation tables using parallel processing, such as *Eval16BitSbox*, require a long time. We used bitslice, which can efficiently process bitwise operations in parallel by taking advantage of independent operations, for generating a linear approximation table. In this study, the linear approximation table generation method implemented using the element unit operation of the existing S-box was upgraded to a *vector* unit operation in a bitslice manner. This improved method enabled the immediate creation of tables, even for 16-bit S-boxes. This approach allows cryptographic algorithm designers to consider a wider variety of S-boxes.

INDEX TERMS Large size S-box, bitsliced implementation, linear approximation table.

I. INTRODUCTION

For post-quantum security, block ciphers with large block sizes are required. If the block size of the block cipher is large, but small-size components are used, a large number of round functions must be used to ensure adequate security. The complexity in one round may increase if the component size is increased. Therefore, large-size components are required for secure and efficient encryption while reducing rounds [1]–[5]. However, the security measurement becomes more complicated as the size of components increases. A substitution box (S-box) is a representative nonlinear function used in cryptographic algorithms.

The S-box has various security properties, including differential uniformity, linearity (or non-linearity), algebraic degree, and strict avalanche criteria. When an S-box is used for an encryption algorithm, cryptanalysis is usually performed using these security properties of the S-box.

The associate editor coordinating the review of this manuscript and approving it for publication was Chao Shen.

Differential cryptanalysis and linear cryptanalysis are the most significant and powerful attack techniques [6]–[11]. Differential cryptanalysis is an analysis technique that exploits the effect of differences in the inputs and outputs of each round. When using an S-box for a nonlinear function, this attack uses a difference distribution table (DDT) of the S-box. The DDT is a chart of how often a difference in the input bit makes a difference in the output bit. Linear cryptanalysis uses an equation generated by a linear approximation of the relationship between the input and output bits of the S-box. Such an approximation can be achieved using a linear approximation table (LAT) of the S-box.

Cryptanalysis largely utilizes these two tables. Finding an S-box for which both properties are secure is therefore an important issue for cryptographic algorithm designers. S-box analysis tools, such as PEIGEN, SAGE, SET, BSAT, etc., generally provide facilities for the generation of DDT and LAT [12]–[15]. However, as the bit size of the S-box grows to around 16 bits, most tools do not support LAT generation, or the generation process is not finished. In the most widely used table generation method for an n-bit S-box, 2^{2n} computations are required to generate the DDT, and 2^{3n} computations are required to generate the LAT. Hence, for a 16-bit S-box, the DDT requires 2^{32} computations, and the LAT requires 2^{48} computations. The amount of computation necessary for LAT generation is difficult to perform on a typical computer. To the best of our knowledge, papers proposing a 16-bit S-box suggested a theoretical security boundary or used *Eval16BitSbox* [1]–[3], [5]. For example, in the case of MISTY, the boundary of security was presented through the structural proof of the S-box, but a concrete security was not presented. A tool, called *Eval16BitSbox*, generates the LAT of 16-bit S-boxes using parallel processes [16]. Using a parallel programming, the amount of resources used (the number of workers) can speed up the generation of the LAT, but for most computers, the computational resources required are still too large, at 2^{3n} divided by the number of workers. To the best of our knowledge, there is no tool that can perform this amount of computation and generates an LAT for a 16-bit S-box in a short time.

We searched for a universal and easy way to generate LATs for large S-boxes, and found a solution in bitslice. A feature of *bitslice* proposed by Kwan, is that it can efficiently parallelize bitwise operations [17]. This feature is often used to effectively implement cryptographic algorithms [18]–[20]. It is also used to process parallel S-boxes in mobile and embedded platforms because of its efficient parallel processing and constant operation speed [21], [22]. A characteristic feature of algorithms using bitslice is that independent operations are grouped into registers for effective parallel processing. In this study, we applied bitslice to LAT generation. We computed the LATs and the linear spectrums for 16-bit S-boxes. This approach solved the problem of the large amount of computation required to generate LATs, and could effectively generate a 16-bit LAT in a few seconds.

The contributions of this study are as follows:

1) Our proposed high-speed LAT generation method performs operations in units of *vector* by bitslice. The amount of computation is reduced by pre-computation. Our proposed tool reduces the n-bit S-box LAT generation time to approximately $\mathcal{O}(n^2)$ from $\mathcal{O}(n^3)$. The resulting 16-bit S-box LAT is generated in less than 3s.
2) We generated 8- and 16-bit S-box LATs based on our method and others and compared them. We found that the LAT generation time of an S-box in a single core was 65–1,200 times faster than that of existing algorithms.
3) For the first time, we presented the whole of the linear spectrum and the linearity of MISTY's 16-bit S-box based on our improved LAT generation method.
4) We expect our method to help cryptographic algorithm designers to create a wider variety of S-boxes.

II. PREVIOUS LINEAR APPROXIMATION TABLE GENERATION METHODS AND GENERATION SPEED LIMITATIONS

The LAT of an S-box is a table that lists the linear biases between the input and output masks. For input Λa and output Λb masks, the LAT of the n-bit S-box (S_n) is defined as follows:

$$LAT(\Lambda a, \Lambda b) = \{x \in \mathbb{Z}_2^n | x \cdot \Lambda a = S_n(x) \cdot \Lambda b\} - 2^{n-1} \quad \text{for } \Lambda a, \Lambda b \in \mathbb{Z}_2^n.$$

The maximum absolute value of the LAT, excluding the (0, 0) entry, is called linearity. The count of the number of biases in the LAT is a linear spectrum. Low linearity and a small number of high biases in the linear spectrum make linear cryptanalysis difficult. Algorithm 1 is the most basic LAT generation algorithm known, based on the abovementioned definition [12]–[15].

Algorithm 1 Basic Algorithm for Generating LAT

```
Require: bit size $n$, $S_n$
1: for $\Lambda a$ in range($2^n$) do
2:     for $\Lambda b$ in range($2^n$) do
3:         LAT[$\Lambda a$][$\Lambda b$] $= -2^{n-1}$
4:         for x in range($2^n$) do
5:             LAT[$\Lambda a$][$\Lambda b$]$+= (x \cdot \Lambda a) \oplus (S_n[x] \cdot \Lambda b)$
6:         end for
7:     end for
8: end for
9: return LAT
```

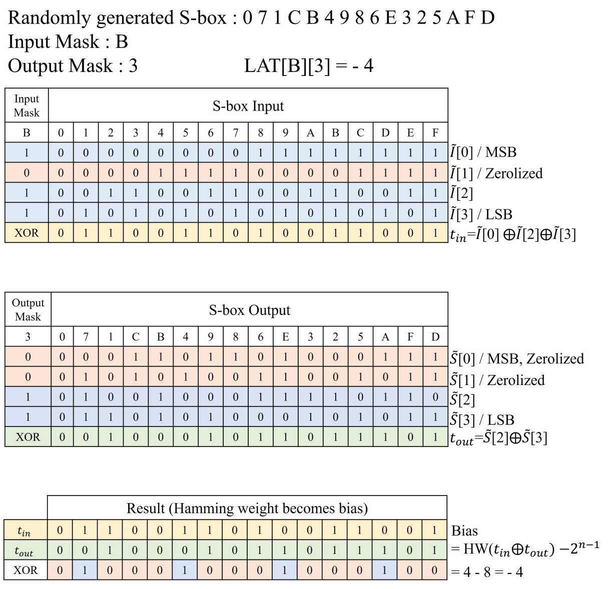
A rough calculation based on the dot product of input and output masking indicates that the basic algorithm requires an operation of $\mathcal{O}(n^3)$ for an n-bit S-box. The previous LAT generation algorithm requires an additional eight times operation whenever the S-box size increased by 1 bit; hence, it is not suitable for the LAT generation of large S-boxes, such as 16-bit S-boxes. *Eval16BitSbox* processes these algorithms in parallel to improve the speed of execution.

III. NEW METHOD FOR GENERATING LINEAR APPROXIMATION TABLES WITH A BITSLICED IMPLEMENTATION

A. APPLICATION OF THE KEY IDEA

The application of bitslicing to an n-bit S-box S is to see the S-box as an n-tuple of component functions $S = (S_0, S_1, \cdots, S_{n-1})$ and encode the Boolean functions $S_i : GF(2^n) \rightarrow GF(2)$ to the truth table of its every outputs as a 2^n-bit vector [23], [24]. For example, the identity permutation I over $GF(2^4)$ would be represented as the 4-tuple of 16-bit words $(0 \times 00FF, 0 \times 0F0F, 0 \times 3333, 0 \times 5555)$. In this paper, we denote the bitsliced f as $\widetilde{f}$.

Line 4 of Algorithm 1 can be operated in parallel, because it independently computes x for the masks, Λa and Λb. The key concept of our study is to compute line 4 using bitslice. First, we changed the $x \cdot \Lambda a$ bitslice representation and



Randomly generated S-box : 0 7 1 C B 4 9 8 6 E 3 2 5 A F D
Input Mask : B
Output Mask : 3 LAT[B][3] = - 4

Input Mask	S-box Input																
B	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	$\tilde{I}[0]$ / MSB
0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	$\tilde{I}[1]$ / Zerolized
1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	$\tilde{I}[2]$
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	$\tilde{I}[3]$ / LSB
XOR	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	$t_{in}=\tilde{I}[0] \oplus \tilde{I}[2] \oplus \tilde{I}[3]$

Output Mask	S-box Output																
3	0	7	1	C	B	4	9	8	6	E	3	2	5	A	F	D	
0	0	0	0	1	1	0	1	1	0	1	0	0	0	1	1	1	$\tilde{S}[0]$ / MSB, Zerolized
0	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	1	$\tilde{S}[1]$ / Zerolized
1	0	1	0	0	1	0	0	0	1	1	1	1	0	1	1	0	$\tilde{S}[2]$
1	0	1	1	0	1	0	1	0	0	0	1	0	1	0	1	1	$\tilde{S}[3]$ / LSB
XOR	0	0	1	0	0	0	1	0	1	1	0	1	1	1	0	1	$t_{out}=\tilde{S}[2] \oplus \tilde{S}[3]$

	Result (Hamming weight becomes bias)																
t_{in}	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	Bias
t_{out}	0	0	1	0	0	0	1	0	1	1	0	1	1	1	0	1	$= HW(t_{in} \oplus t_{out}) - 2^{n-1}$
XOR	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	= 4 - 8 = - 4

FIGURE 1. LAT calculation example.

considered "x" "$I(x)$". Next, we obtained the value t_{in} by XORing the components corresponding to Λa. Subsequently, we calculated $\widetilde{S}$ and acquired the t_{out} value by XORing the components corresponding to Λb. Finally, the bias for masks Λa and Λb was computed as the Hamming weight (HW) by XORing t_{in} and t_{out}. Fig. 1 shows an example. In the data, the top is the most significant bit (MSB), and the bottom is the least significant bit (LSB).

Algorithm 2 is a non-optimized bitsliced implementation of the LAT, reconstructed based on the abovementioned methodology. $\widetilde{N}$ indicates bitsliced data N. I is the identity permutation of $GF(2^n)$ indicating S_n input.

Algorithm 2 Generating LAT With Bitsliced Implementation(1)

```
Require: bit size $n$, $\widetilde{S_n}$
1: for $\Lambda a$ in range($2^n$) do
2:     for $\Lambda b$ in range($2^n$) do
3:         $t_{in} = 0$; $t_{out} = 0$
4:         for $k$ in range($n$) do
5:             $t_{in} = t_{in} \oplus (\widetilde{I}[k] * ((\Lambda a \gg k) \& 1))$
6:             $t_{out} = t_{out} \oplus (\widetilde{S_n}[k] * ((\Lambda b \gg k) \& 1))$
7:         end for
8:         LAT[$\Lambda a$][$\Lambda b$] = $HW(t_{in} \oplus t_{out}) - 2^{n-1}$
9:     end for
10: end for /* HW returns hamming weight */
11: return LAT
```

B. ALGORITHM IMPROVEMENT

Algorithm 2 has room for improvement with respect to speed. We analyzed the LAT generation process and improved the speed by reducing the loops and pre-computing the t_{in} and t_{out} operations.

1) PRE-COMPUTATION PROCESSES

To improve the performance, we pre-computed and tabulated the data generated by S-box independent operations before the LAT generation process. The S-box dependent operations were also pre-computed when the LAT generation process was started. With these pre-computations, the main operations (lines 5 and 6 in Algorithm 2) in the LAT generation were changed to $\mathcal{O}(1)$ and $\mathcal{O}(n)$, respectively, requiring only simple table references and HW calculations. The generation algorithm of the LAT was divided into three operations: t_{in} generation; t_{out} generation; and an XOR of the results. The first operation, t_{in}, is dependent only on $\widetilde{I}$ and Λa. We can produce the table M_{in} by pre-computating it before generating the LAT. The amount of computation can be reduced by making and using M_{in} for each S-box size. The second operation, t_{out}, does not have any dependency on Λb. We can remove t_{out} from the loop and generate table M_{out} by pre-computation, as with t_{in}. For the final operation, the innermost loop becomes two table references and a simple XOR.

2) SEPARATION OF THE LAT MASK

We separated the mask to reduce the amount of computation involved in M_{in}, M_{out} and the storage space of all tables by half. In this process, each loop was reduced by half, but the innermost operation was quadrupled. We considered each group when the MSB of the n-bit mask was 0 and 1. The groups were identical, except for the MSB. Thus, when the MSB of the mask was 0, the LAT value was computed, and the LAT value, including the MSB, was naturally computed by XORing the I[0] S[0] corresponding to the MSB. For example, in the case of an 8-bit S-box S_8, for the two masks Λa and $\Lambda b < 128$, we first computed $LAT[i][j] = HW(\text{some } t) - 128$ according to line 8 of Algorithm 2. We then computed the following terms for each Λa and Λb:

$$LAT[\Lambda a|128][\Lambda b] = HW(\text{some } t \oplus \widetilde{I}[0]) - 128$$
$$LAT[\Lambda a][\Lambda b|128] = HW(\text{some } t \oplus \widetilde{S_8}[0]) - 128$$
$$LAT[\Lambda a|128][\Lambda b|128] = HW(\text{some } t \oplus \widetilde{I}[0] \oplus \widetilde{S_8}[0]) - 128$$

This procedure cuts the memory required and the amount of computation needed for the pre-computation by more than half. Algorithm 4 is an improved bitsliced implementation for LAT generation.

A rough calculation based on the innermost loop, suggests that the improved algorithm requires $\mathcal{O}(n^2)$ operations for an n-bit S-box.

IV. COMPARISON WITH PREVIOUS METHODS

We investigated various tools used to analyze the S-box properties, to evaluate the performance of the proposed algorithm. We used a personal computer with an INTEL core i7-11700K Processor@ 3.6 GHz, NVIDIA GTX 1080ti, 64 GB RAM, and MSI MAG Z590 TORPEDO mainboard

Algorithm 3 Pre-Computation Process

```
Require: bit size $n$
 1: for $\Lambda a$ in range($2^{n-1}$) do
 2:     $t = 0$
 3:     for $k$ in range($n$) do
 4:         $IN[\Lambda a][k] = (\Lambda a \gg k)\&1$
 5:         $t = t \oplus (\widetilde{I}[k] * IN[\Lambda a][k])$
 6:     end for
 7:     $M_{in}[\Lambda a] = t$
 8: end for
 9: return $M_{in}, IN$
```

Algorithm 4 Generating LAT With Bitsliced Implementation(2)

```
Require: bit size $n$, $\widetilde{S_n}$, $M_{in}$, $IN$ (from algorithm 3)
 1: for $\Lambda b$ in range($2^{n-1}$) do
 2:     $t = 0$
 3:     for $k$ in range($1, n$) do
 4:         $t = t \oplus (\widetilde{S_n}[k] * IN[\Lambda b][k])$
 5:     end for
 6:     $M_{out}[\Lambda b] = t$
 7: end for
 8: $IS = \widetilde{I}[0] \oplus \widetilde{S_n}[0]$
 9: for $\Lambda a$ in range($2^{n-1}$) do
10:     for $\Lambda b$ in range($2^{n-1}$) do
11:         $m = M_{in}[\Lambda a] \oplus M_{out}[\Lambda b]$
12:         $LAT[\Lambda a][\Lambda b] = HW(m) - 2^{n-1}$
13:         $LAT[\Lambda a|2^{n-1}][\Lambda b] = HW(m \oplus \widetilde{I}[0]) - 2^{n-1}$
14:         $LAT[\Lambda a][\Lambda b|2^{n-1}] = HW(m \oplus \widetilde{S_n}[0]) - 2^{n-1}$
15:         $LAT[\Lambda a|2^{n-1}][\Lambda b|2^{n-1}] = HW(m \oplus IS) - 2^{n-1}$
16:     end for
17: end for /* $HW$ returns hamming weight */
18: return LAT
```

for our experiments. For the 8-bit S-boxes, we collected cases in which the source code was released together with an article. We excluded cases in which the tool did not work properly, or in which the LAT generation result was incorrect. The tool language was divided into C/C++ or Python, but the LAT generation tool implemented in Python was treated as C/C++ because it was implemented as *ctypes*, a C compatible library. Thus, for Python, the basic algorithm was implemented and compared. Finally, we selected the S-box analysis tools *PEIGEN* [12], *SET* [13], and *SAGE* [14] as the LAT generation time comparison groups for the 8-bit S-boxes. We measured the average of the LAT generation times of 10,000 random 8-bit S-boxes using each tool, except for the basic algorithm implemented in Python. This basic algorithm was used to measure only 100 random 8-bit S-boxes, because of the problem of execution time. In the case of 16-bit S-boxes, the LAT generation time was compared with that of Eval16BitSbox, an LAT generator that supports multi-core-based parallel processes through the Parallel Java 2 Library [25]. Due to the generation speed problem,

TABLE 1. Comparison of LAT generation speed with existing tools and our study.

ine 8-bit S-box	Code	C/C++				Python	
	Ref.	PEIGEN [12]	SAGE [14]	SET [13]	Ours	Basic	Ours
	Time	12ms	171ms	42ms	0.182ms	>1m	50ms
ine 16-bit S-box	Code	JAVA		C		CUDA	
	Ref.	*Eval16BitSbox* [16]		Ours		Ours	
	Speed	262h/[workers]		1h30m/[workers]		<3s	

Eval16BitSbox estimated the time based on the bias generation time of the output masks corresponding to the random Λas at randomly generated 16-bit S-boxes. Table 1 shows the experimental results.

Among the tools implemented for an 8-bit or less S-box in Table 1, PEIGEN was the best performing tool based on the C/C++ language, taking 12 milliseconds to generate an LAT. The slowest tool was SAGE, because it utilized the C library in python. We also tested other tools shared on GitHub, as well as the tools in Table 1, but we could not find any tool with a better performance than PEIGEN. Our method took 182 microseconds to generate the LAT, an over 65× performance improvement. In the case of Python, the LAT generation method implemented using Algorithm 1 required more than 1 minute. The rapid generation of LATs in Python requires the use of a C library through ctypes, such as SAGE. However, our method, implemented in Python, took 50 milliseconds to generate an LAT, an over 1,200× performance improvement. When generating LATs for a 16-bit S-box, the performance was improved by over 174×. The notation [workers] in Table 1 means the threads used, and the time was measured based on a single thread. Our method was effective even with a single thread, but parallel programming it is was more effective, due to the use of bitslice. As a result of parallel programming using CUDA, it took up to 3 seconds to generate an LAT for a 16-bit S-box. PEIGEN, SAGE and SET did not work for the LAT generation of a 16-bit S-box due to the large size of the S-box.

V. CONCLUSION

Our study has widened the size of S-boxes that can be investigated in the future, and it is expected to be valuable to designers of cryptographic algorithms using large S-boxes.

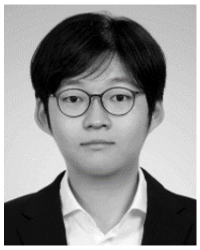
In this study, we developed a method of generating LATs using a bitsliced representation. The amount of computation was reduced to approximately 2^{2n} by pre-computing t_{in} and t_{out}. Our algorithm was faster than other tools, and the LAT of a 16-bit S-box could effectively be generated in a short time. Our study widened the size of S-boxes that can be investigated in the future. We expect our method to be valuable to cryptographic algorithm designers using large S-boxes.

REFERENCES

[1] M. Kelly, A. Kaminsky, M. Kurdziel, M. Lukowiak, and S. Radziszowski, "Customizable sponge-based authenticated encryption using 16-bit S-boxes," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Oct. 2015, pp. 43–48.
[2] S. M. Komissarov, "On algorithmic implementation of 16-bit S-bo-XES with ARX and butterfly structures," *Prikladnaya Diskretnaya Matematika. Prilozhenie*, vol. 5, no. 12, pp. 101–107, Sep. 2019.

[3] P. Barreto, V. Nikov, S. Nikova, V. Rijmen, and E. Tischhauser, "Whirlwind: A new cryptographic hash function," *Designs, Codes Cryptogr.*, vol. 56, nos. 2–3, pp. 141–162, Aug. 2010.
[4] A. Canteaut, S. Duval, G. Leurent, M. Naya-Plasencia, L. Perrin, T. Pornin, and A. Schrottenloher, "Saturnin: A suite of lightweight symmetric algorithms for post-quantum security," *IACR Trans. Symmetric Cryptol.*, pp. 160–207, Jun. 2020.
[5] M. Matsui, "New block encryption algorithm misty," in *Proc. Int. Workshop Fast Softw. Encryption*, Springer, 1997, pp. 54–68.
[6] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *J. Cryptol.*, vol. 4, no. 1, pp. 3–72, 1991.
[7] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Heidelberg, Germany: Springer, 2012.
[8] M. Matsui, "Linear cryptanalysis method for des cipher," in *Proc. Workshop Theory Appl. Cryptograph. Techn.* Heidelberg, Germany: Springer, 1993, pp. 386–397.
[9] J. Daemen and V. Rijmen, *The Design of Rijndael*, vol. 2. New York, NY, USA: Springer-Verlag, 2002.
[10] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, FL, USA: CRC Press, 2020.
[11] H. M. Heys, "A tutorial on linear and differential cryptanalysis," *Cryptologia*, vol. 26, no. 3, pp. 189–221, Jul. 2002.
[12] Z. Bao, J. Guo, S. Ling, and Y. Sasaki, "PEIGEN—A platform for evaluation, implementation, and generation of S-boxes," *IACR Trans. Symmetric Cryptol.*, vol. 2019, pp. 330–394, Mar. 2019.
[13] S. Picek, L. Batina, D. Jakobović, B. Ege, and M. Golub, "S-box, set, match: A toolbox for S-box analysis," in *Proc. IFIP Int. Workshop Inf. Secur. Theory Pract.* Heidelberg, Germany: Springer, 2014, pp. 140–149.
[14] *Sagemath—Open-Source Mathematical Software System*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.sagemath.org/
[15] P. K. Behera and S. Gangopadhyay, "BSAT: A new tool for analyzing cryptographic strength of Boolean function and S-box of symmetric cryptosystem," in *Progress in Advanced Computing and Intelligent Engineering*. Singapore: Springer, 2021, pp. 557–569.
[16] A. Kaminsky. *Block Cipher Analysis*. Accessed: Oct. 20, 2021. [Online]. Available: http://cs.rit.edu/~ark/parallelcrypto/blockcipheranalysis/
[17] M. Kwan, "Reducing the gate count of bitslice des," *IACR Cryptol. ePrint Arch.*, vol. 2000, no. 51, p. 51, 2000.
[18] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES implementation: A high-throughput bitsliced approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2211–2222, Oct. 2019.
[19] N. Nishikawa, H. Amano, and K. Iwai, "Implementation of bitsliced AES encryption on CUDA-enabled GPU," in *Proc. Int. Conf. Netw. Syst. Secur.* Cham, Switzerland: Springer, 2017, pp. 273–287.
[20] V. Grosso, G. Leurent, F. Standaert, and K. Varici, "LS-designs: Bitslice encryption for efficient masked software implementations," in *Fast Software Encryption* (Lecture Notes in Computer Science), vol. 8540. Heidelberg, Germany: Springer, 2014, pp. 18–37.
[21] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 4727, P. Paillier and I. Verbauwhede, Eds. Heidelberg, Germany: Springer, 2007, pp. 450–466.
[22] H. Kim, Y. Jeon, G. Kim, J. Kim, B.-Y. Sim, D.-G. Han, H. Seo, S. Kim, S. Hong, J. Sung, and D. Hong, "PIPO: A lightweight block cipher with efficient higher-order masking software implementations," in *Int. Conf. Inf. Secur. Cryptol.* Cham, Switzerland: Springer, 2020, pp. 99–122.
[23] J. Jean, T. Peyrin, S. M. Sim, and J. Tourteaux, "Optimizing implementations of lightweight building blocks," *IACR Trans. Symmetric Cryptol.*, vol. 2017, pp. 130–168, Dec. 2017.
[24] E. Biham, "A fast new des implementation in software," in *Proc. Int. Workshop Fast Softw. Encryption*. Heidelberg, Germany: Springer, 1997, pp. 260–272.
[25] A. Kaminsky. *Parallel Java 2 Library*. Accessed: Oct. 20, 2021. [Online]. Available: https://cs.rit.edu/ ark/pj2.shtml

GIYOON KIM received the B.S. degree in information security, cryptology and mathematics from Kookmin University, Seoul, South Korea, in 2019, where he is currently pursuing the dual M.S. and Ph.D. degree in financial information security. His research interests include cryptographic primitives, cryptanalysis, artificial intelligence, and digital forensics.



YONGJIN JEON received the B.S. degree in mathematics and the M.S. degree in financial information security from Kookmin University, Seoul, South Korea, in 2018 and 2020, respectively, where he is currently pursuing the Ph.D. degree in financial information security. His research interests include cryptographic primitives, cryptanalysis, and symmetric cryptosystems.

JONGSUNG KIM received the B.S., and M.S. degrees in mathematics from Korea University, South Korea, in 2000, and 2002, respectively. He received double Doctoral degrees on "Combined Differential, Linear and Related-Key Attacks on Block Ciphers and MAC Algorithms", completed in 2006, and 2007 at the ESAT/COSIC group of Katholieke Universiteit Leuven, Belgium and at Engineering in Information Security of Korea University, respectively. Dr. Kim is Full Professor of Departments of Information Security, Cryptology and Mathematics & of Financial Information Security, Kookmin University, South Korea. His research interests include cryptanalysis, symmetric cryptosystems, and digital forensics.