IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 16, 2019, accepted July 20, 2019, date of publication July 25, 2019, date of current version August 13, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2930986

SCA-Resistant GCM Implementation on 8-Bit AVR Microcontrollers

SEOG CHUNG SEO[1], (Member, IEEE), AND HEESEOK KIM[2]
[1]Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, South Korea
[2]Department of Cyber Security, Korea University, Sejong 30019, South Korea

Corresponding author: HeeSeok Kim (80khs@korea.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) through the Korea Government (MSIT) under Grant 2019R1F1A1058494.

ABSTRACT Galois/counter mode (GCM) is one of the most widely used authenticated encryptions. To date, even though some works have investigated the security against side channel analysis (SCA) in the process of GCM computation, especially GHASH function, they failed to present comprehensive SCA security in consideration of both SPA/TA and DPA/CPA aspects simultaneously. In this paper, we present a secure GCM implementation on 8-bit AVR microcontroller environments. The proposed implementation provides comprehensive SCA security in consideration of not only SPA/TA but also DPA/CPA. In order to defeat SPA/TA, we introduce the concepts of dummy XOR with garbage registers and instruction level atomicity (*ILA*) and also present secure binary field (*BF*) multiplication method using them, which runs in a constant-time and fixed pattern. We also propose an efficient multiplicative masking method which can prevent DPA/CPA when computing GHASH function in the GCM process. Through actual implementation of the proposed method on an 8-bit AVR ATmega128 microcontroller, we show that the proposed method outperforms existing alternatives while providing comprehensive SCA security. With respect to the performance of secure binary field multiplication, the proposed multiplication method outperforms the related work by around 51.86% when computing a 128-bit binary field multiplication. Regarding the overhead of the multiplicative masking method, the proposed method requires only one additional *BF* multiplication and negligible amount of field additions regardless of the number of input blocks, while the related work consumes around the $\{\log(m+n+1)+2\}$ number of additional *BF* multiplications when there are $(m+n+1)$ input blocks. Through SCA-related experiments, we prove the SCA security of the proposed methods.

INDEX TERMS Secure binary field multiplication, Galois/counter mode (GCM) mode, masking, side channel analysis (SCA), authenticated encryption (AE), simple power analysis (SPA), timing analysis (TA), differential power analysis (DPA), correlation power analysis (CPA).

I. INTRODUCTION

Authenticated encryptions (AEs) have been widely used in secure communications where confidentiality and integrity are required at the same time. Recently, AEs have gained popularity on even resource-constrained platforms including smart cards, sensor nodes, and RFIDs since they can provide both confidentiality and integrity within a single scheme rather than combining two separate encryption method and authentication method.

AES-GCM (Galois/Counter mode of operation), proposed by McGrew and Viega [1], [2] and standardized by NIST [3] in 2007, is one of the most widely used AEs in practical applications such as SSL/TLS, SSH, IPSec, IEEE 802.1 AE (MACsec), NSA suite B, IEEE 802.11ad, and so on. It consists of AES encryption in counter mode as well as authentication tag generation based on GHASH function involving multiplications in $GF(2^{128})$. While AES-GCM is computationally secure against existing cryptanalyses, several works conducting side channel analysis (SCA) [4] on GHASH function on GCM mode (especially targeting field multiplication of $GF(2^{128})$ in GHASH function) have recently been presented [5]–[7]. Since its demand is increasing for embedded devices, and since such devices are deployed in the field which can be easily captured by adversaries, secure countermeasures against SCA are fundamental requirements

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

for the implementation of AES-GCM on embedded devices.

To date, there have been a number of studies devoted to analyzing SCA security on AES primitive itself as well as developing efficient and secure countermeasures. On the other hand, in recent years, researchers have realized the importance of SCA security on GHASH function (especially binary field multiplication in $GF(2^{128})$) and presented several attack methods [5]–[11]. In addition to attack methods, some countermeasures have been also proposed for securing GHASH of GCM. For example, Liu *et al*. proposed the constant-time polynomial multiplication method based on the Karatsuba technique and Masked Block-Comb (*BC*) method [10], [11]. Although they achieved TA (Timing Analysis) resistance when computing binary field (*BF*) multiplications over $GF(2^{128})$ in the GHASH function, they did not consider DPA/CPA [4], [12] security on the overall process of GHASH function (DPA and CPA denote Differential Power Analysis and Correlation Power Analysis, respectively). Most recently, Oshida *et al*. proposed a multiplicative masking method for defeating DPA/CPA-type SCA when computing the GHASH function [9]. Even though they achieved complete DPA/CPA security through the use of their proposed multiplicative masking method, they ignored the SPA/TA security of the underlying *BF* multiplications (SPA and TA denote Simple Power Analysis and Timing Analysis, respectively). Furthermore, their scheme requires a large number of field multiplications over $GF(2^{128})$ in order to unmask the final result before outputting it (this is called the correctness property).

In this paper, we concentrate on developing efficient SCA-resistant GCM implementation for 8-bit AVR microcontrollers which are widely used for sensor nodes, smart cards, and RFIDs. First, we review and analyze the security of the aforementioned existing SCA countermeasures [9]–[11] on the GHASH function of AES-GCM with respect to comprehensive SCA security. Secondly, we introduce concepts of Dummy XOR with garbage registers and instruction level atomicity (*ILA*) and present a new SPA/TA-resistant *BF* multiplication method based on these concepts for 8-bit AVR microcontrollers. Thirdly, we propose a multiplicative masking method for defending DPA/CPA on GHASH function. The proposed method is superior to Oshida *et al*'s method in terms of computational overhead. Furthermore, we show through experimentation that the proposed masked GHASH function using the proposed *BF* multiplication method is secure against SPA/TA and DPA/CPA.

The contributions of this work can be summarized as follows.

- *Analyzing the weakness of the related works*
  We revisit the SPA security of Liu *et al*.'s *BF* multiplication method [10], [11] and show that their method is vulnerable to SPA, in contrast to their assertion. Through analyzing the SPA security of the *BF* multiplication methods used in the GHASH function and the DPA security of the process of GHASH itself, we show that considering comprehensive security regarding SPA/TA and DPA/CPA for secure GHASH function is necessary.
- *Introducing concepts of Dummy XOR with garbage registers and instruction level atomicity (ILA) and developing an SPA/TA-resistant BF multiplication method for 8-bit AVR microcontrollers*
  The proposed *BF* multiplication method is based on the Block-Comb (*BC*) method and the Karatsuba technique. In order to make the *BC* method secure against TA, we introduce instruction level atomicity (*ILA*), which can supplement the timing difference incurred from the use of conditional branch and jump instructions in the *BC* method. We also propose Dummy XOR with garbage registers in order to prevent SPA, and this makes it difficult for attackers to determine whether the actual multiplier bit is 0 or 1 while observing a power consumption trace in SPA.
- *Presenting a multiplicative masked GHASH function secure against DPA/CPA analysis*
  For defending DPA/CPA in the process of GHASH function, we present a masked GHASH function. In contrast to Oshida *et al*.'s method [9] requiring around the $\{log(m+n+1)+2\}$ number of additional *BF* multiplications for correctness property, our method requires roughly one field addition operation per each input block as well as only one *BF* multiplication, regardless of the number of input blocks. Thus, our method can save around the number of $\{log(m+n+1)+1\}$ *BF* multiplications compared with Oshida *et al*.'s method.
- *Implementing the proposed BF multiplication method and masked GHASH function on an 8-bit ATmega128 microcontroller and analyzing their security with actual experiments*
  Through actual implementation on an 8-bit AVR ATmega128 microcontroller, we show that the proposed *BF* multiplication method is at least 51.86% faster than Liu *et al*.'s method while providing SPA/TA security. We also show through SCA experiments that the proposed methods are secure against SPA/TA and DPA/CPA.

The remainder of this paper is organized as follows. Section II briefly reviews the GCM process, binary field multiplication, and its implementation methods on the 8-bit AVR environment. Section III describes the SCA target points of GHASH function and criteria for secure GHASH implementation, and also describes the existing countermeasures and their weaknesses. Section IV describes the proposed countermeasures including a secure *BF* multiplication method and a multiplicative masked GHASH function. Section V presents the implementation results and security analysis with actual experiments. Finally, section VI concludes this paper with suggestions for future works.

II. RELATED WORKS

In this section, we briefly review GCM mode which is our target implementation and identify which information

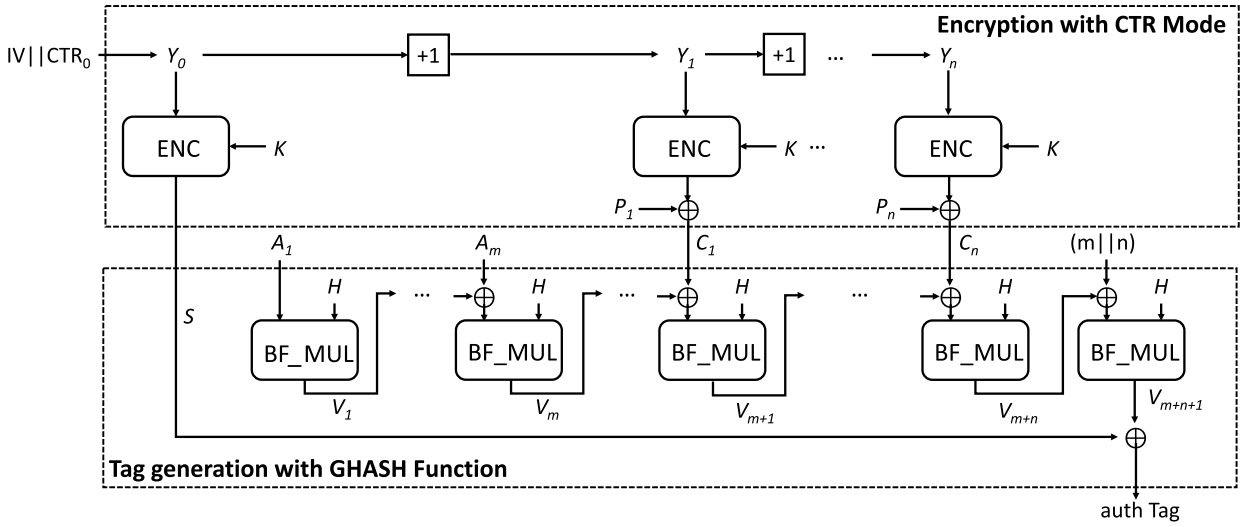


FIGURE 1. Overview of GCM process (Consisting of CTR encryption mode and GHASH function) [1], [3], [8], [9].

needs to be protected against SCA. Then, we introduce 8-bit AVR microcontroller with respect to the number of registers, memory size, and instruction set since our software is developed on this architecture. Finally, we describe the existing binary field multiplication methods on 8-bit AVR platforms. There are mainly two kinds of multiplication methods: LUT (LookUp Table)-based methods and Block-Comb (BC)-based methods.

A. BRIEF REVIEW OF GCM MODE

GCM (Galois/Counter Mode) mode is one of the most widely used authenticated encryption methods which simultaneously provides data confidentiality and authentication within a single scheme [1]–[3]. It employs the CTR mode for encryption and the GHASH function for authentication tag generation. GHASH function uses binary field (BF) multiplication over $GF(2^{128})$ with an irreducible polynomial $f(z) = z^{128} + z^7 + z^2 + z + 1$. Fig. 1 depicts the process of GCM mode. In the figure, `ENC` and `BF_MUL` refer to encryption and 128-bit BF multiplication, respectively. For each block of message, CTR encryption mode in GCM encrypts the counter and conducts bit-wise XORs the encrypted result with the corresponding message block to produce ciphertext block; the counter value is then incremented. The ciphertext is then bit-wise XORed with an accumulator of the GHASH function part in GCM, which is then multiplied with a hash key H, where $H = ENC(K, 0^{128})$, in $GF(2^{128})$. Note that if the associated data is provided, the concatenation of the associated data and the generated ciphertext is the input of the GHASH function. The concatenated input is divided into 128-bit blocks, and each block is sequentially multiplied with H. Since the BF multiplication over $GF(2^{128})$ is the core part of the GHASH function in GCM, it needs to be implemented both efficiently and securely.

In the process of GCM, not only must the secret key K for encryption to be protected, but so must the hash key H and the subkey S.[1] If either H or S is exposed, attackers can compute valid (i.e., forged) authentication tags for any pair of associated data and ciphertexts [9], [13]. Thus, it is necessary to protect both H and S along with K in the process of GCM. The aim of this paper is to develop an efficient and secure countermeasure which can protect H and S against SCA.

B. 8-BIT AVR MICROCONTROLLERS

Presently, 8-bit AVR microprocessors are widely used for various applications, such as smartcards, sensor nodes in WSN, RFIDs, and so on. 8-bit AVR microcontrollers, including ATmega128, have 32 general-purpose registers $(R_{31}, \ldots, R_1, R_0)$, with six of them being used for memory address pointers (Each pair of (R_{26}, R_{27}), (R_{28}, R_{29}), and (R_{30}, R_{31}) are aliased as X, Y, and Z pointer registers, respectively) [14]. AVR microprocessors have separate memory areas and buses for program and data in a simple single-issue pipeline, because they are based on the Harvard architecture. They have a total of 133 instructions, and each instruction typically has a fixed latency. For example, the arithmetic/logical instructions (e.g. `ADD` (arithmetic add), `EOR` (bit-wise XOR), `LSL` (logical shift left), and so on) are executed in a single clock cycle, while the memory access instructions (e.g. `LD` (load from memory to register), `ST` (store from register to memory), and so on) take two clock cycles [14]. Note that conditional branch instructions such as `BREQ` (branch if equal), `SBRS` (skip if bit in register is set), and so on consume different numbers of clock cycle(s) depending on whether the condition is true or false.[2] 8-bit AVR microcontrollers have limited computation and memory capabilities. For example,

[1] Until now, even though a huge number of studies have been conducted for securing K during AES encryption, the SCA security of GHASH function has not been fully investigated. Thus, we concentrate on the SCA security of GHASH function in this paper.

[2] For example, in the `SBRS` case, if the condition is false, it consumes 1 cycle. Otherwise, it consumes 2 or 3 cycles depending on the word size of the skipped instruction.

TABLE 1. Existing binary field multiplication methods on 8-bit AVR processor (*ECC* means elliptic curve cryptosystem).

	Technique	Application	Fields	Timing (cc)	SCA Resistance
Seo et al. [16]	4-bit wise LookUp-Table	ECC	$GF(2^{163})$	19,670	SPA, TA
Aranha et al. [17]	4-bit wise LookUp-Table	ECC	$GF(2^{163})$	4,508	SPA, TA
Shirase et al. [19]	Block-Comb	ECC	$GF(2^{233})$	9,511	none
Seo et al. [20]	Unbalanced Block-Comb	ECC	$GF(2^{163})$	4,346	none
Seo et al. [21]	Karatsuba Block-Comb	ECC	$GF(2^{163})$	3,274	none
Seo et al. [21]	Constant Karatsuba Block-Comb	ECC	$GF(2^{163})$	5,005	TA
Seo et al. [22]	Enhanced Karatsuba Block-Comb	ECC	$GF(2^{233})$	6,898	none
Ziu et al. [10], [11]	Masked Block-Comb	GCM	$GF(2^{128})$	14,445	TA

in case of 8-bit ATmega128, it has only 4 Kbytes of RAM and 128 Kbytes of ROM, and it runs at 7.3728 MHz. In contrast to the latest x86 CPUs and ARM processors supporting carryless multipliers, which are generic multipliers for binary field multiplication, AVR microcontrollers still do not contain the dedicated hardware for carryless multiplication.

C. BINARY FIELD MULTIPLICATION ON 8-BIT AVR PLATFORMS

The core operation required for the GHASH function of GCM is the Binary Field (BF) multiplication with an input operand as an associated data block or ciphertext block and a secret constant element H. To date, many studies for optimizing the BF multiplication on 8-bit AVR platforms have been conducted [15]–[21]. They can mainly be divided into two categories: LUT (LookUp Table)-based methods [15]–[18] and Block-Comb (BC)-based methods [19]–[21]. The results of the existing methods are summarized in Table 1 and the detail will be described in Sec. II-C2 and Sec. II-C3. Since the number of available registers is limited on 8-bit AVR (only 26 general purpose registers, except for 6 memory address pointer registers, are available), a number of memory accesses occur when computing a BF multiplication. For example, at least 64 registers are required to hold an entire set of a multiplicand, a multiplier, and a result of the BF multiplication over $GF(2^{128})$. However, since only 26 registers are available, only certain parts of the operands can be maintained in the registers, which results in a number of redundant memory accesses. Thus, the main concern of studies on the BF multiplication on 8-bit AVR microcontrollers is to minimize the number of redundant memory accesses by optimizing the use of the available registers.

Before describing our proposed BF multiplication method, this section briefly describes the existing BF multiplication methods and certain notations on 8-bit AVR platforms.

1) BINARY FIELD MULTIPLICATION AND NOTATIONS

The Binary Field (BF) multiplication involves computing $A \cdot B$ where $A = \sum_{i=0}^{m-1} a_i z^i$, $B = \sum_{i=0}^{m-1} b_i z^i \in GF(2^m)$. A and B refer a multiplicand and a multiplier, respectively, and the result of BF multiplication C can be expressed as $C = \sum_{i=0}^{m-1} A \cdot b_i z^i$. The most basic BF multiplication algorithm is the Shift-and-Add (or Shift-and-Xor) method. This method involves scanning the multiplier from the 0-th bit to the $(m-1)$-th bit. At each iteration, multiplicand A is left-shifted such as $A \cdot z$, and if the bit of multiplier B is set to 1, then the left-shifted multiplicand is XORed with the accumulator (Namely, if b_i, i-th bit of the multiplier, is set 1, then, $A \cdot z^i$ is XORed with the accumulator). The Comb method, the basic algorithm for LUT-based methods and Block-Comb method, improves the performance of the BF multiplication by taking advantage of the fact that if $A \cdot z^k$ has been computed for some $k \in [0, W-1]$ (in cases of 8-bit AVR platforms, W is 8), $A \cdot z^{Wj+k}$ can be easily obtained by appending j zero words to the right of the vector representation of $A \cdot z^k$. Thus, it can reduce the number of shift operations as compared with the basic Shift-and-Add algorithm. There are two types of Comb methods: the LtR Comb method and the RtL Comb method. While the LtR Comb method scans a multiplier from MSB (Most Significant Bit) to LSB (Least Significant Bit), the RtL Comb method proceeds from LSB to MSB [15], [23].

Throughout the paper, we will use the following notations. R_i refers to the i-th general purpose register where $31 \geq i \geq 0$. The operators $\oplus$, $\ll$, and $\gg$ denote XOR, logical left shifts, and logical right shifts. $A[i]$ means the i-th byte (or word) of A and it is composed of eight bits like $(a_{8i+7}, \ldots, a_{8i})$. Finally, $A[i, \ldots, j]$ represents the bytes (words) from $A[j]$ to $A[i]$, respectively.

2) LOOK-UP TABLE METHODS

In order to improve the efficiency of the 128-bit BF multiplication of GCM, McGrew *et al.* described different approaches involving tables of different sizes, allowing for trade-off between memory requirement and computation speed [1], [2] in their GCM specification. Their methods use differently sized-tables of 256 Bytes, 4 Kbytes, 8 Kbytes, and 64 Kbytes, and the performances of the methods were measured on a 32-bit RISC Motorola G4 processor. Even though McGrew *et al.*'s table-based approaches are efficient in terms of computation speed, their memory requirements are too large to be used on 8-bit AVR microcontrollers. Thus, GCM implementations on resource constrained devices as 8-bit AVR and 16-bit MSP430 microcontrollers typically have utilized López *et al.*'s Look-Up Table (LUT)-based approach [11], [24] which was originally proposed for the BF multiplication over binary elliptic curves [15], [23].

López *et al.*'s LUT-based BF multiplication method is an extension of the LtR Comb method (so-called wLtR Comb

method), and it executes the *BF* multiplication by w-bit unit rather than single bit unit at the expense of a precomputation table, which results in a reduced the number of bit operations, such as shift and XOR operations [15], [17], [23], [24]. It first builds a precomputation table involving all possible results of $A \cdot u(z)$ for all polynomials $u(z)$ of degree at most $w-1$. It then scans multiplier B by w-bit unit at a time from MSB to LSB and takes the corresponding value from the precomputation table, and the value is XORed with the intermediate result without computing it. On 8-bit AVR platforms, it is known that 4-bit is the optimal width w for LUT-based method, which requires $16 \times m$-bit of RAM memory for accommodating a table composed of 16 polynomials from $0 \cdot A$ to $(z^3 + z^2 + z + 1) \cdot A$. The LUT-based method has been widely implemented on 8-bit AVR platforms [16]–[18]. For example, Seo *et al*. [16] implemented the 163-bit polynomial multiplication with NesC language on an 8-bit ATmega128 microcontroller, and they improved the original LUT-based method by reducing the number of redundant memory accesses by combining two iterations of the main loop into one. They achieved 21.1% of performance improvement and got 19,670 *cc* for a field multiplication over $GF(2^{163})$. In [17], [18], Aranha *et al*. introduced a rotating register mechanism which could significantly reduce the number of memory accesses required by the LUT-based polynomial multiplication method. They implemented their method in Assembly language and achieved timings of 4,508 *cc*, 8,314 *cc*, and 11,727 *cc* in computing each polynomial multiplication over $GF(2^{163})$, $GF(2^{233})$, and $GF(2^{271})$, respectively (*cc* means clock cycle). While LUT-based methods provide good performance and are secure against TA and SPA, they are inherently vulnerable to side channel analysis (SCA) using memory-address information [11], [25] due to the huge number of memory accesses they involve. In [10], [11], Liu *et al*. [11] successfully attacked López *et al*.'s LUT-based *BF* method with a kind of horizontal correlation analysis [26]. They could find the LUT index by using the correlation between power consumption patterns for constructing the LUT and accessing the LUT during the actual multiplication.

3) BLOCK-COMB METHOD

As an alternative to LUT-based methods, the Block-Comb (*BC*) method was first introduced in [19] for the efficient *BF* multiplication of η_T pairing computation on the 8-bit ATmega128 microcontroller. In the *BC* method, a multiplier and a multiplicand are divided into equal-sized blocks of s-bytes, and partial products of multiplicand blocks and multiplier blocks are computed in a column-wise fashion. The partial product is then computed using the Comb method for efficiency. In other words, in the *BC* method, the available registers are divided into three parts: s registers for a block-sized multiplicand, s registers for a block-sized multiplier, and $2s + 1$ for the result of partial products. Since the intermediate results are maintained in $2s + 1$ working registers, the results of partial products belonging to the same column can be directly updated to the registers without needing to access memory, which reduces the number of redundant memory accesses. In [19], Shirase *et al*. concluded that the optimal block size s is 6 based on $(4s + 1) < 26$. The original *BC* computes a polynomial multiplication over $GF(2^{239})$ within 9,511 clock cycles (*cc*).

Seo *et al*. [20] extended the size of the block from 6 to 7 in proposing the Unbalanced Block-Comb method (*UBC*) for $GF(2^{163})$ multiplication. They took advantage of the fact that the tested bits of a multiplier are no longer necessary during the process of a partial product, and recycled this register for holding the most significant byte of the multiplicand. As a result, the extended block size reduces the number of partial products from 16 to 9 when computing a polynomial multiplication over $GF(2^{163})$ (Note that 7-word (resp. 6-word) block size divides 163-bit polynomial into three blocks (resp. four blocks)). *UBC* could compute a field multiplication over $GF(2^{163})$ with $4,546$. Then, Seo *et al.* [21] proposed the Karatsuba Block-Comb method (*KBC*), developed by combining the Karatsuba technique with Block-Comb, which reduces the number of partial products further from 9 to 6 at the expense of several cheap field additions when computing a polynomial multiplication over $GF(2^{163})$. *KBC* could achieve 3,274 *cc* for a field multiplication over $GF(2^{163})$. They also presented a variant of constant-time Karatsuba block comb method. Even though it achieved timing attack (TA) resistance, it is still vulnerable to simple power analysis (SPA). Most recently, Seo *et al*. [22] presented an enhanced Karatsuba Block-Comb (*EKBC*) method by proposing novel multiplier encoding technique which can significantly reduce the number of required registers for maintaining the multiplier. They achieved a new speed record for elliptic curve scalar multiplication over NIST-compliant K-233 curve. Until now, *EKBC* provides the fastest computation timing for binary field multiplication. In 2018, Seo *et al.* [10], Liu *et al.* [11] presented a secure GCM implementation on 8-bit AVR processor. For SPA and TA resistance, they presented a masked Block-Comb method. However, in contrast to their assertion, their method is still vulnerable to SPA. We will show the SPA weakness of their method in Sec. III in detail. In summary, since *BC*-based methods do not use a Look-Up Table, they are secure against a kind of horizontal CPA used for attacking López *et al*.'s LUT-based method.

III. SIDE-CHANNEL ATTACK ON GHASH OF GCM

Recently presented attacks have mainly focused on the *BF* multiplication in the GHASH function [5]–[11] and their goal has been to recover the secret hash key H in order to forge the authentication tag. Among the aforementioned studies, [10], [11] considered SPA/TA security of *BF* multiplication primitive itself and [5]–[7], [9] investigated the DPA/CPA security of the *BF* multiplication in the process of GHASH.

A. SCA TARGET POINTS AND CRITERIA FOR SECURE GCM IMPLEMENTATION

From the above-mentioned studies, the SCA target points to the GHASH function can be summarized as follows.

- `Target of SPA/TA`
  If $A \cdot H$, a BF (Binary Field) multiplication in GHASH, is computed with an algorithm in which its execution path depends on the multiplier, it leaks information regarding the multiplier. For example, if the Shift-and-Add method is used for a BF multiplication, as in [9], the multiplicand is bit-wise XORed on the accumulator when the tested bit of the multiplier is set to 1. In other words, the timing differences and power consumption pattern during the execution of the method can be exploited by attackers.
  $\Rightarrow$ *Therefore, the BF multiplication in GHASH needs to both execute in constant-time and generate a regular power consumption pattern regardless of the bit value of the multiplier.*
- `Target of DPA/CPA`
  One of the inputs to the BF multiplications in GHASH is a known value, as associated data A or ciphertext C, while the other input is fixed and secret hash key H. Therefore, attackers can apply DPA/CPA-type attacks during the process of GHASH function in order to find the hash key H.
  $\Rightarrow$ *Therefore, a proper masking method needs to be applied so as to randomize the known input value in order to prevent these kinds of attacks. Furthermore, the intermediate result in GHASH needs to be randomized until the generation of the final result.*

B. SECURITY OF THE EXISTING COUNTERMEASURES

Regarding the secure implementation of GCM, several works have been conducted [9]–[11]. In this subsection, we examine the claimed SCA security of the existing methods.

Seo *et al.* [10] and Liu *et al*. [11] proposed the constant-time BF multiplication based on the Karatsuba technique and the Block-Comb (BC) method. In their paper, they showed that López *et al*.'s LUT-based BF multiplication method is vulnerable to a kind of horizontal CPA [26] exploiting its inherent consecutive memory accesses, even though it is known to be secure against TA and SPA. The BC-based BF multiplication methods are known to be secure against this kind of SCA because they do not use Look-Up Tables. However, the BC-based methods are vulnerable to Timing Attack (TA) and Simple Power Analysis (SPA), because the partial products in the BC-based methods are computed using either the LtR Comb method or the RtL Comb method. In other words, in both the LtR Comb method and the RtL Comb method, the multiplicand is XORed with the accumulator only when the tested bit value of the multiplier is set to 1. Thus, the power consumption pattern or timing difference during the computation of the BC-based methods can reveal the value of the multiplier. In order to defend TA, *NOP* instruction or XORing with zero register can be used.[3] However, even though both approaches make BC method run in constant-time, Liu *et al*. pointed out that the routines using *NOP* instruction or XORing with zero register consume relatively lower power consumption than normal XOR operations conducted when the tested bit is set to 1 due to the different hamming weights and instructions, which results in being vulnerable to SPA [11]. Therefore, in order to prevent not only SPA/TA but also SCA exploiting correlation between memory access patterns and hamming weights of operands, they eventually proposed a Masked Block-Comb (MBC) method which runs in constant-time and always executes the same execution pattern, regardless of the value of multiplier.

Algorithm 1 Masked Block-Comb on 32-bit [10], [11]

Require: 32-bit wise operands A and B
Ensure: Result C=$A \cdot B$

```
for l = 7 to 0 do
  for m = 3 to 0 do
    BIT ← A[m]&(1 ≪ l)
    MASK, T0 ← (0 − BIT)
    for k = 3 to 0 do
      C[k + m] ← C[k + m] ⊕ (B[k]&MASK)
    end for
  end for
  C ← C ≪ 1
end for
(Return C)
```

Alg. 1 shows the masked BC method from [10], [11]. However, this algorithm does not show a regular power consumption pattern in contrast to their assertion, so it does not guarantee SPA resistance. In the case in which the tested bit is set (resp. clear), BIT becomes 1 (resp. 0), then $0xFF$ (resp. $0x00$) is assigned to $MASK$. Consequently, ($B[k]\&0xFF$) (resp. ($B[k]\&0x00$)) is bit-wise XORed with the accumulator $C[k + m]$ when the tested bit is set (resp. clear). Since ($B[k]\&0x00$) is the same as $0x00$, this is identical to XORing the accumulator with zero register. Therefore, we found that the power consumption pattern of Alg. 1 presented in [10], [11] is similar to that using XORing with zero register. Furthermore, Liu *et al*. did not consider DPA/CPA security on the overall process of the GHASH function.

Recently, Oshida *et al*. proposed the first DPA/CPA countermeasure based on multiplicative masking for the purpose of securing the GHASH function [9] against the previously-presented DPA scenarios [5]–[7]. Fig. 2 shows Oshida *et al*.'s masked GHASH function. First, the input block (A_1) is masked with ($M \oplus S$), where M is a 128-bit random masking value and S is the subkey. Then, the masked input value as ($A_1 \oplus M \oplus S$) is multiplied with the hash key H, and the result of the multiplication is XORed with ($S \oplus S \cdot H$). Note that

[3] When computing BF multiplication, if the tested bit is zero, *NOP* instruction or XORing the accumulator with zero register can be used to hide the time difference.

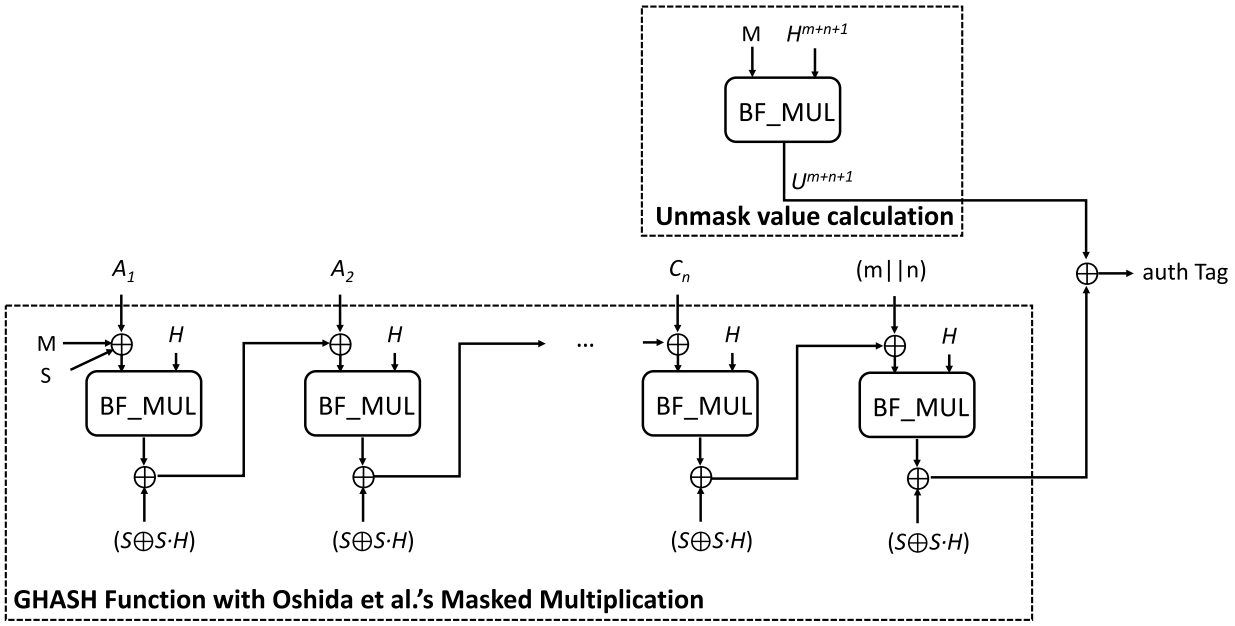


FIGURE 2. Oshida *et al.*'s masked GHASH function.

by XORing the output of i-th BF_MUL with $(S \oplus S \cdot H)$, the updated result always becomes masked with $(M \cdot H^i \oplus S)$. Although they achieved complete DPA/CPA security, they ignored SPA/TA security on the BF multiplication method used in the GHASH function. Since they used the Shift-and-Add method as described in Alg. 6 in the Appendix for computing BF multiplications in GHASH, the timing difference or power consumption pattern during the multiplications can leak the secret information. Furthermore, their scheme requires a large number of field multiplications over $GF(2^{128})$ in order to unmask the final result before outputting it (this is called the correctness property). In other words, it requires the computation of $(M \cdot H^{m+n+1})$ in order to unmask the final result for the correctness property, which requires $(log(m+n+1)+2)$ binary field multiplications. Oshida *et al.*' mentioned that the value of H^{m+n+1} can be precomputed prior to conducting the GHASH function. However, the precomputed H^{m+n+1} can be reused only when the number of input blocks is $(m+n+1)$. For example, if H^{10} is precomputed, it can only be used without any additional cost when the number of input blocks on the GHASH function is 10. However, since the number of input blocks in the GHASH function is not practically fixed, precomputing H^{m+n+1} does not always provide a computational advantage, in contrast to their assertion.

In Section V, we analyze the security of Liu *et al.*'s Masked Block-Comb (MBC) method (Alg. 1) and the Shift-and-Add method (Alg. 6) used in Oshida *et al.*'s masked GHASH function in terms of SPA. Regarding Liu *et al.*'s method, we show that the power consumption pattern of their MBC method is the same as that of XORing with the zero register method. In addition, we show that the power consumption pattern of the Shift-and-Add method depends on the secret information. Then, we also show that the MBC method does not guarantee DPA/CPA security.

IV. PROPOSED SECURE GCM COMPUTATION

In this section, we describe the proposed techniques for a secure GHASH function in terms of SPA/TA and DPA/CPA. In Section III, we define design criteria for the secure implementation of GHASH in order to achieve comprehensive SCA security. According to the design criteria, first, the BF multiplication in GHASH needs to not only execute in constant-time but also generate a regular power consumption pattern. Secondly, the known input value such as associated data or ciphertext needs to be randomized with a proper masking value so as to block attackers from correctly guessing the intermediate result of GHASH.

A. PROPOSED BINARY FIELD MULTIPLICATION

In a manner identical to the Masked Block Comb method presented in [10], [11], we basically make use of the 32-bit wise Block-Comb (BC) method. In order to compute a BF multiplication over $GF(2^{128})$, each operand of the multiplication is divided into four blocks, and each partial product of the divided multiplicands and multipliers is computed using the 32-bit wise RtL Comb method. Alg. 2 computes a partial product of a 32-bit multiplicand and a 32-bit multiplier in an RtL Comb fashion in the BC-based BF multiplication method. In Alg. 2, the multiplicand A is bit-wise XORed with the intermediate result C when the tested bit of the multiplier B is set to 1. Otherwise, no operation is performed. Thus, it is vulnerable to TA and SPA. The first goal of this paper is to convert Alg. 2 into a secure version, which not

Algorithm 2 Block Comb Method on 32-Bit Where $(R_7, \ldots, R_0)$, $(R_{12}, \ldots, R_8)$, and $(R_{16}, \ldots, R_{13})$ Are Reserved for Accumulator, Multiplicand, and Multiplier

Require: 32-bit wise operands A (a multiplicand) and B (a multiplier)
Ensure: Result $C(64\text{-}bit) = A \cdot B$

for $l = 0$ to 7 **do**
  $R_l \leftarrow 0$
end for
for $l = 0$ to 3 **do**
  $R_{8+l} \leftarrow A[l]$
  $R_{13+l} \leftarrow B[l]$
end for
$R_{12} \leftarrow 0$
for $l = 0$ to 7 **do**
  for $m = 0$ to 3 **do**
    if the l-th bit of $R_{13+m} == 1$ **then**
      for $k = 0$ to 4 **do**
        $R_{m+k} \leftarrow R_{m+k} \oplus R_{8+k}$
      end for
    end if
  end for
  if $l \neq 7$ **then**
    $(R_{12}, \ldots, R_8) \leftarrow (R_{12}, \ldots, R_8) \ll 1$
  end if
end for
(Return C)

only executes in a constant time but also generates a regular power consumption pattern. In order to defend TA and SPA, we introduce the concepts of Dummy XOR with the garbage registers and Instruction Level Atomicity (*ILA*), then apply them to the *BC* method.

1) DUMMY XOR WITH GARBAGE REGISTERS FOR REGULAR POWER CONSUMPTION PATTERN

In Section III, we show that the power consumption pattern in Liu *et al.*'s MBC method depends on the value of the tested multiplier bit even though identical operations are executed regardless of the bit value (Section V presents the experimental results regarding the SPA security on Liu *et al.*'s method). This is because zero values ($(B[k]\&MASK)$) are XORed with the accumulator C when the tested value is zero, which is identical to the method of XORing with a zero register. Thus, we introduce the concept of Dummy XOR with the garbage registers to prevent SPA. We double the number of registers for the accumulator C from $(R_7, \ldots, R_0)$ to $(R_{15}, \ldots, R_0)$. Our method using the double-sized accumulator completely makes use of the 25 registers for computing a partial product of 32-bit operands, and this is acceptable in 8-bit AVR microcontrollers, where 32 registers are available. Figure 3 shows the register assignment in the proposed method. The set of $(R_0, \ldots, R_7)$ plays the garbage registers and the set of $(R_8, \ldots, R_{15})$ contains the real intermediate result of the partial products. Our method executes XOR operations not with zero values, but with the real multiplicand regardless of the value of tested bit; however, the destination of the operations needs to be correctly configured. For example, when the tested bit is 0, the $(R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k})$ is computed where $k = 0$ to 4. Otherwise, $(R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k})$ is computed. Since the real multiplicand is XORed with the accumulator in both cases, the power consumption patterns of both cases are indistinguishable with SPA.

2) INSTRUCTION LEVEL ATOMICITY FOR CONSTANT-TIME EXECUTION

If we implement the proposed Dummy XOR using the garbage registers with `if-then-else` statements, this leaks the timing difference. For example, the naive Dummy XOR method can be implemented as following.

if the l-th bit of $R_{21+m} == 1$ **then**
  for $k = 0$ to 4 **do**
    $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
  end for
else
  for $k = 0$ to 4 **do**
    $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
  end for
end if

Even though the above codes execute the same operations as $(R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k})$ or $(R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k})$ regardless of the tested bit value, it leaks the timing difference. This is because of the `if-then-else` conditional branches. On the 8-bit AVR architecture, the conditional branch instructions inherently consume different clock cycles based on whether or not the condition is true. For example, if the condition is false (resp. true), they usually consume 1 clock cycle (resp. 2 clock cycles). Thus, even though the inner operation patterns in the `if` and `else` clauses are

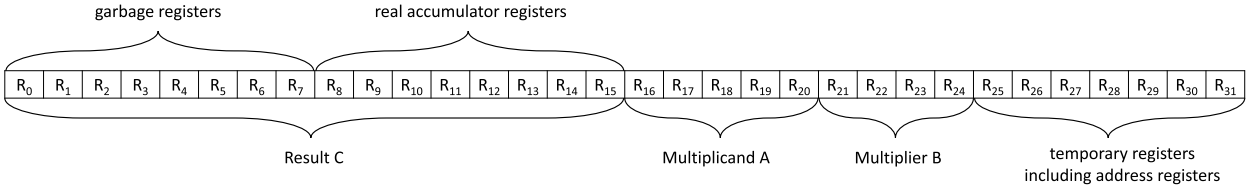


FIGURE 3. Register assignment in the proposed block-comb (*BC*) method.

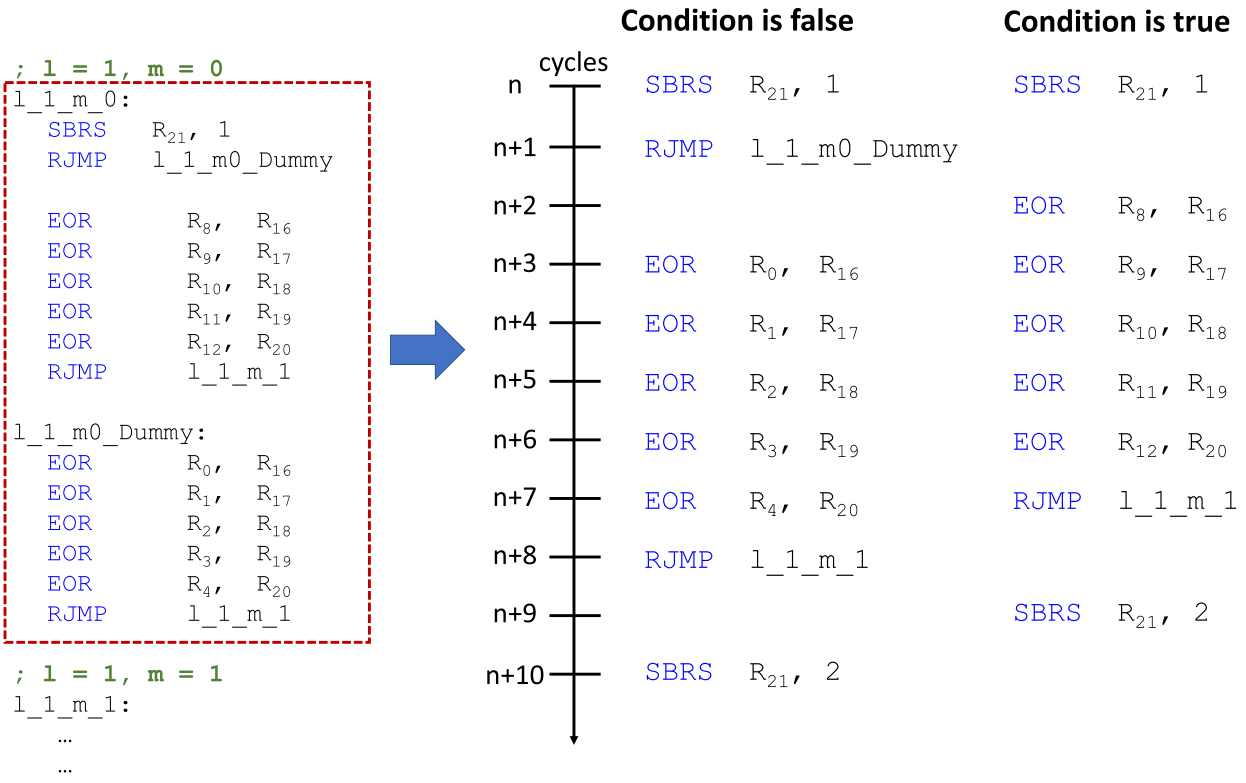


FIGURE 4. Example of the naive implementation of Dummy XOR method with the garbage registers when the 1-st bit of the 0-th byte in the multiplier is tested (Left figure shows assembly codes for implementing Dummy XOR with the garbage registers and right figure shows the clock cycle counts for two cases: When the condition is either false or true).

identical, it can incur a timing difference, which results in vulnerability to TA. Thus, we need to address this problem to develop a constant-time *BF* multiplication method which is secure against TA.

Fig. 4 shows the assembly level implementation of the Dummy XOR method with the garbage registers and its execution time in clock cycles. The conditional branch in the *BC* method is implemented with an SBRS (Skip if bit in register is set) instruction [14], [19]. As shown in the figure, the core part of the *BC* method using the Dummy XOR technique is implemented with the SBRS, RJMP, and EOR instructions. SBRS tests whether or not the 1-st bit of R_{21} is set. If the tested bit is false, SBRS consumes 1 clock cycle and the next instruction is executed. For example, when the tested bit is clear, the program counter (PC).[4] jumps to the label of *l*_1_*m*0_*Dummy* with RJMP and then the multiplicand is XORed with the garbage registers through the five Dummy EOR instructions. When the tested bit is set, SBRS consumes 2 clock cycles and the next instruction is skipped. Thus, the multiplicand is XORed with the real accumulator registers through the five EOR instructions. Thus, the naive implementation of the Dummy XOR technique is vulnerable to TA. Furthermore, the timing difference makes it so that the two conditional cases are distinguishable from each other with SPA. Therefore, we need to ensure that the two conditional cases use the same kinds of instructions at the same time.

We define the concept of Instruction Level Atomicity (*ILA*), which makes it so that the two execution paths use the same kinds of instructions at the same time. In order to design instruction level atomicity, we analyze the internal processes of the SBRS and RJMP instructions in detail. The SBRS (Skip if Bit in Register is Set) instruction increments the program counter (PC) by 1 (resp. 2) when the condition is false (resp. true). In other words, since the PC is incremented by 1 when the condition is false, the next instruction is executed. On the other hand, when the condition is true, the PC is updated as PC+2, which results in executing the second instruction by skipping the first instruction from SBRS. Thus, the main work of SBRS is to increment the PC according to the result of the testing condition. When updating the PC, the adder in ALU is used. In other words, when the condition is false (resp. true), the adder is executed once (resp. twice), which requires 1 clock cycle (resp. 2 clock cycles). Therefore, it can be found that the execution of the SBRS instruction is the same as adding 1 to the PC register once or twice with the ADD instruction. The RJMP (Relative Jump) instruction updates the PC by PC+k + 1 where k is the displacement, and it takes 2 clock cycles for its execution. RJMP also uses the adder in ALU twice for updating the PC as $PC \leftarrow PC + 1$ and $PC \leftarrow PC + k$. Based on the above

[4]PC is a register to hold/store the address of the next instruction to be executed by the microcontroller's microprocessor.

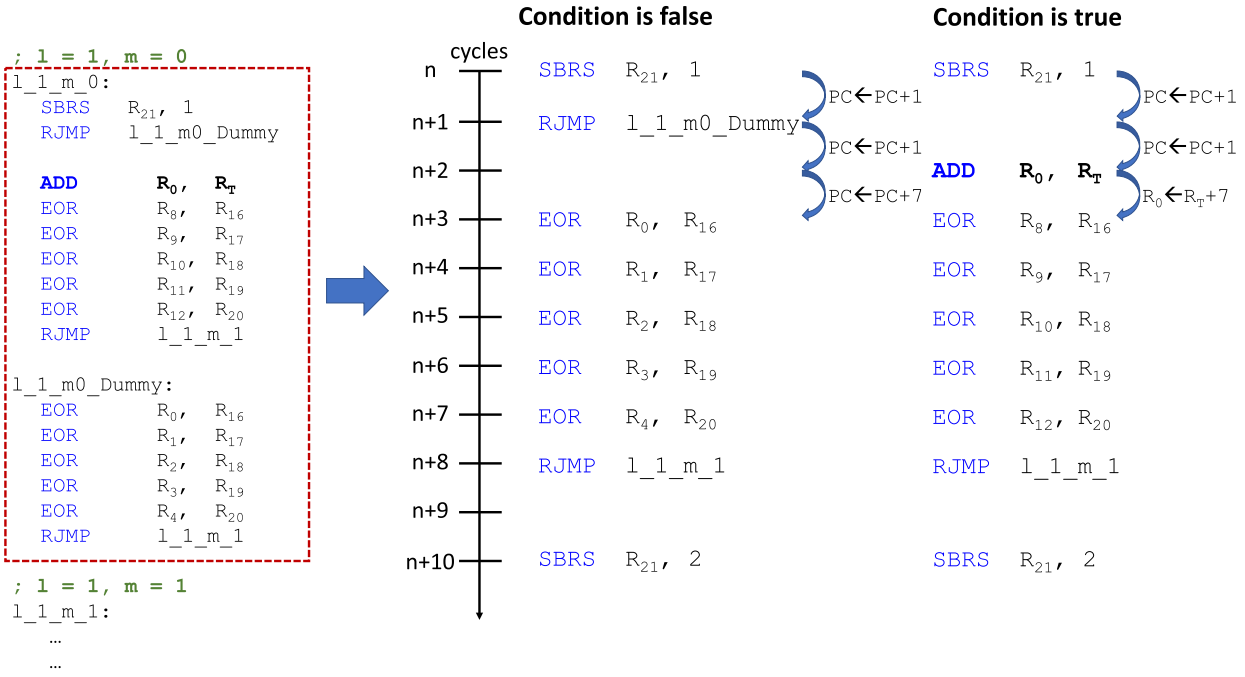


FIGURE 5. Example of the application of instruction level atomicity when $l = 1$ and $m = 0$ (left figure shows the assembly codes for constant-time dummy XOR with the garbage registers and right figure shows the cycle counts for two cases: The condition is false or true).

analysis, we can conclude that the primary operation of the SBRS and RJMP instructions is to update the PC with the adder in ALU, which is identical to the execution of the ADD instruction. Therefore, we make use of the ADD instruction in order to make two execution cases (the case when the condition is false and the case when the condition is true) indistinguishable from each other. Fig. 5 shows the proposed instruction level atomicity for making two execution cases indistinguishable with respect to the types of instructions and execution cycles. Since the displacement k is seven in the proposed method, R_T register holding $0x07$ is added to R_0 register belonging to the garbage registers (In the proposed method, R_{25} can be used as R_T). We determine that two execution cases consume the same number of clock cycles as 10, and the type of instruction at each clock cycle is identical. Thus, our method is secure against TA and SPA.

Alg. 3 shows the proposed *BC* method on 32-bit operands by combining the proposed Dummy XOR with the garbage registers and *ILA*. Alg. 3 executes a partial product of 32-bit operands, and is secure against TA and SPA through the application of Dummy XOR using the garbage registers and instruction level atomicity using a dummy addition. If the tested multiplier bit is set (that is, if the condition is true), the multiplicand is XORed with the real accumulator consisting of registers $(R_{15}, \ldots, R_8)$ after the dummy add instruction is executed. Otherwise (the condition is false), Dummy XOR is executed by XORing the multiplicand with the garbage registers $(R_7, \ldots, R_0)$. The core parts in Alg. 3 as steps 10–25 and steps 26–38 are implemented in a loop unrolling manner as codes depicted in Fig. 5.

3) APPLICATION OF KARATSUBA TECHNIQUE

Since the proposed *BC* method computes a partial product of 32-bit operands, the 128-bit *BF* multiplication requires 16 partial products. Thus, we apply the Karatsuba technique so as to reduce the number of partial products for efficiency. The Karatsuba technique allows us to reduce the number of partial products in sub-quadratic complexity with a few additional field additions [27], [28]. Rather than directly applying the Karatsuba technique, we apply multiple levels of the Karatsuba technique: 1-level Karatsuba for computing a 64-bit multiplication requiring three 32-bit partial products and 2-level Karatsuba for computing a 128-bit multiplication requiring three 64-bit partial products. Thus, the proposed 128-bit *BF* multiplication requires nine 32-bit wise partial products, rather than sixteen in total.

Alg. 4 shows the proposed 1-level Karatsuba *BC* method for 64-bit operands, and it requires three 32-bit wise partial products (denoted as $\times_{32-bit}$) which are computed using Alg. 3. Note that the shift operations of $(H \ll 64)$ and $(M \ll 32)$ at step 5 can be efficiently computed by arranging the byte positions rather than using the bit-shift operations. The 2-level Karatsuba *BC* method for computing a 128-bit *BF* multiplication can be constituted in a manner similar to Alg. 4 with the difference being that the size of operands increases from 64-bit to 128-bit. The 2-level Karatsuba *BC* method requires three 64-bit partial products and the partial products are computed with Alg. 4.

We have implemented the proposed *BC* methods on an 8-bit AVR ATmega128 microcontroller running on 7.3728 MHz. Table 2 shows the timing costs of the proposed

Algorithm 3 Proposed Block-Comb (BC) Method on 32-Bit Where $(R_{15}, \ldots, R_0)$, $(R_{20}, \ldots, R_{16})$, and $(R_{24}, \ldots, R_{21})$ Are Reserved for Accumulator, Multiplicand, and Multiplier

Require: 32-bit multiplicand A and 32-bit multiplier B
Ensure: Result C(64-bit)$= A \cdot B$ (Among sets of $R_{15}, \ldots, R_0$ for holding the result C, the sets of $(R_7, \ldots, R_0)$ and $(R_{15}, \ldots, R_8)$ are used to hold the garbage result and real result, respectively)

$R_{25} \leftarrow 0x07$ // Set displacement value for dummy `ADD` instruction for *ILA*
for $l = 0$ to 15 **do**
  $R_l \leftarrow 0$
end for
for $l = 0$ to 3 **do**
  $R_{16+l} \leftarrow A[l]$
  $R_{21+l} \leftarrow B[l]$
end for
$R_{20} \leftarrow 0$
// Processing from 0-th bit to 6-th bit
for $l = 0$ to 6 **do**
  for $m = 0$ to 3 **do**
    if the l-th bit of $R_{21+m} == 1$ **then**
      $R_0 \leftarrow R_0 + R_{25}$ // Dummy `ADD` instruction for *ILA*
      for $k = 0$ to 4 **do**
        $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
      end for
    else
      // Dummy `XOR` with the garbage registers
      for $k = 0$ to 4 **do**
        $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
      end for
    end if
  end for
  $(R_{20}, \ldots, R_{16}) \leftarrow (R_{20}, \ldots, R_{16}) \ll 1$
end for
// Processing the final 7-th bit
for $m = 0$ to 3 **do**
  if the 7-th bit of $R_{21+m} == 1$ **then**
    $R_0 \leftarrow R_0 + R_{25}$ // Dummy `ADD` instruction for *ILA*
    for $k = 0$ to 4 **do**
      $R_{8+m+k} \leftarrow R_{8+m+k} \oplus R_{16+k}$
    end for
  else
    // Dummy `XOR` with the garbage registers
    for $k = 0$ to 4 **do**
      $R_{m+k} \leftarrow R_{m+k} \oplus R_{16+k}$
    end for
  end if
end for
(Return $C = (R_{15}, \ldots, R_8)$)

32-bit (Alg. 3), 64-bit (Alg. 4), and the 128-bit wise BF multiplications. We have implemented the proposed 32-bit and 64-bit wise BF multiplications in AVR assembly language.

Algorithm 4 64-Bit Wise Karatsuba Block Comb KAT_MUL_{64-bit}

Require: 64-bit wise operands A and B.
Ensure: Result $C(128\text{-}bit) = A \cdot B$.

$L \leftarrow A[3 \ldots 0] \times_{32-bit} B[3 \ldots 0]$
$H \leftarrow A[7 \ldots 4] \times_{32-bit} B[7 \ldots 4]$
$M \leftarrow (A[7 \ldots 4] \oplus A[3 \ldots 0]) \times_{32-bit} (B[7 \ldots 4] \oplus B[3 \ldots 0])$
$M \leftarrow H \oplus L \oplus M$
$C \leftarrow (H \ll 64) \oplus (M \ll 32) \oplus L$
(Return C)

The 64-bit BF multiplication includes codes of three 32-bit wise BF multiplications and other operations rather than calling the function of the 32-bit wise BF multiplication. The 128-bit BF multiplication is implemented in C language, and it internally calls the 64-bit wise BF multiplication implemented in AVR assembly language.

4) REDUCTION AND BIT REFLECTION

The GCM standard specifies that the bits of the state are reflected [1], [2]. That is, in the GCM standard, with respect to an element over $GF(2^{128})$, the leftmost bit is regarded as the 0-th bit and the rightmost bit is regarded as the 127-th bit, while general cryptographic algorithms usually use the opposite notation. We utilize the table-based bit-reflection for the inputs and the output of a BF multiplication (The table for bit-reflection requires 256-byte). The bit-reflection table converts an input byte into a bit-reflected byte as $(b_0b_1b_2b_3b_4b_5b_6b_7) \leftarrow (b_7b_6b_5b_4b_3b_2b_1b_0)$, and vice versa. The bit-reflection is required for each BF multiplication of the GHASH function. In the process of the GHASH function, the result of each BF multiplication needs to be reduced by the irreducible polynomial $f(z) = z^{128} + z^7 + z^2 + z + 1$. We have implemented a fast reduction method similar to the fast reduction methods in [23], computing byte-unit reduction rather than bit-unit, and using $f(z) = z^{128} + z^7 + z^2 + z + 1$ for 8-bit AVR microcontrollers in C language.

B. MASKED GHASH COMPUTATION

While the proposed BF multiplication method is secure against TA and SPA, it is still vulnerable to DPA/CPA. Thus, this section describes our approach to preventing DPA/CPA. In order to defend DPA/CPA, the power consumption leakage needs to be independent from the processed data (i.e. intermediate results) during the crypto operations. In detail, if a constant secret value as a subkey or a hash key is mixed with varying inputs as blocks of associated data or blocks of ciphertext, DPA/CPA can be launched at the GHASH function. Thus, we propose a multiplicative masking approach with which to randomize both the inputs and the intermediate results during the process of the GHASH function. Fig. 6 shows the process of the proposed masked GHASH function. At the first BF multiplication, the input A_1 is XORed

TABLE 2. Costs for the proposed SPA/TA-resistant 32-bit, 64-bit, and 128-bit wise *BF* multiplication (The cost includes the overhead for function calls such as `POP` and `PUSH` instructions).

Bit	Method	Timing (cc)
32-bit	Proposed Block-Comb	490
64-bit	Proposed Karatsuba Block-Comb (Level 1)	1,330
128-bit	Proposed Karatsuba Block-Comb (Level 2)	5,675

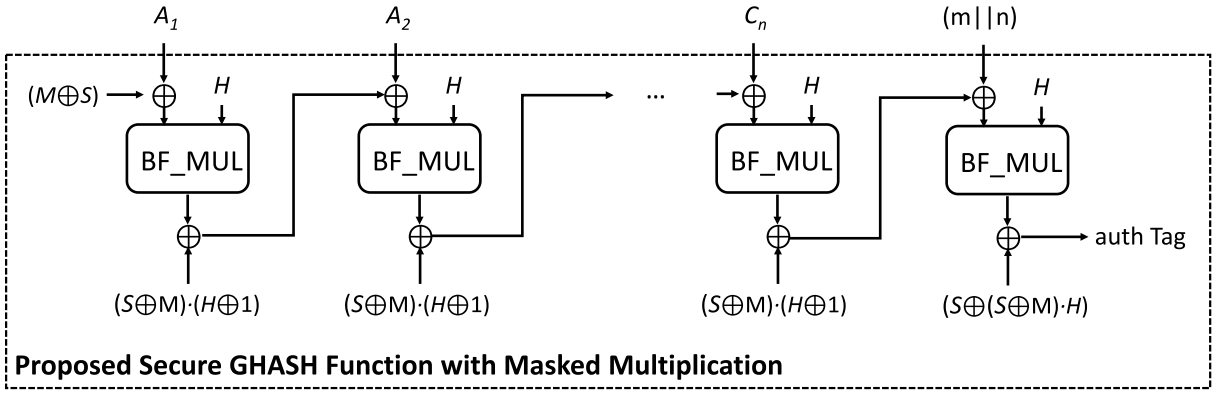


FIGURE 6. Proposed Masked GHASH function process (*M*, *S*, and *H* denote a 128-bit random masking value, a subkey derived from encrypting ($IV||CTR_0$) with a master key *K*, and a hash key derived from encrypting 0^{128} with the master key *K*. `BF_MUL` denotes a 128-bit *BF* multiplication).

with $(M \oplus S)$, where M is a 128-bit random masking value and S is a subkey derived from encrypting $(IV||CTR_0)$ with a master key K. Note that a random value M needs to be newly generated at the beginning of the GHASH function. Since the input value is masked with a random value, the power consumption becomes independent from the processed data of `BF_MUL`, which makes it so that attackers cannot correctly guess the intermediate value. The output of `BF_MUL` is XORed with $(S \oplus M) \cdot (H \oplus 1)$. Since the output of the first `BF_MUL` is $((M \oplus S \oplus A_1) \cdot H)$, the result of $((S \oplus M) \cdot (H \oplus 1)) \oplus ((M \oplus S \oplus A_1) \cdot H)$ becomes $(A_1 \cdot H \oplus M \oplus S)$. In the original GHASH function without any DPA/CPA countermeasures, the output of the first *BF* multiplication is $(A_1 \cdot H)$. As a result, it can be found that the original output of `BF_MUL` is XORed with $(M \oplus S)$. Then, the masked output of the first `BF_MUL` is XORed with the input (A_2) of the second `BF_MUL`, which once again randomizes the input of `BF_MUL`. This process is iterated until each of the $(m + n + 1)$ blocks is computed by `BF_MUL`. As shown in Fig. 6, both the input and the output of `BF_MUL` are masked with random masking values, which guarantees resistance against DPA/CPA during the process of the GHASH function. The output of the final `BF_MUL` needs to be XORed with $(S \oplus S \cdot H \oplus M \cdot H)$ rather than $(S \oplus M) \cdot (H \oplus 1)$ for correctness property.

Alg. 5 shows the detailed execution process of the proposed GHASH function. The input consists of $m + n + 1$ segments (segments mean blocks in this context): m segments of the additional associated data, n segments of ciphertext, and a segment of $(m||n)$. Whenever this GHASH function is invoked, a new 128-bit random value needs to be generated at step 1, which is then XORed with the subkey S at step 2.

Algorithm 5 Proposed Masked GHASH Function

Require: $m + n + 1$ input data segments $D_1, \ldots, D_{m+n}$ (First m segments are additional associated data such as $(D_1 = A_1, \ldots, D_m = A_m)$ and the next n segments are ciphertext such as $(D_{m+1} = C_1, \ldots, D_{m+n} = C_n$. The final segment is $(m||n)$). A subkey S and a hash key H.

Ensure: Authentication Tag T

1: Generate a 128-bit random value M
2: Compute $T \leftarrow M \oplus S$
3: Compute $U \leftarrow$ `BF_MUL`(T, H)
4: Compute $V \leftarrow U \oplus T$ // $V = (S \oplus S \cdot H \oplus M \oplus M \cdot H)$
5: Compute $U \leftarrow U \oplus S$ // $U = (S \oplus S \cdot H \oplus M \cdot H)$
// Processing $m+n$ input segments
6: **for** $i = 1$ to $m + n$ **do**
7: $T \leftarrow T \oplus D_i$
8: $T \leftarrow$ `BF_MUL`(T, H)
9: $T \leftarrow T \oplus V$
10: **end for**
// Processing the final segment
11: $T \leftarrow T \oplus D_{m+n+1}$
12: $T \leftarrow$ `BF_MUL`(T, H)
13: $T \leftarrow T \oplus U$
14: Return (T)

Steps 3–5 compute V and U where V will be XORed with the output of each `BF_MUL` except for the final `BF_MUL` and U will be XORed with the final output of the GHASH function. Steps 6–10 process $m+n$ input segments by calling `BF_MUL`

TABLE 3. Comparison of additional operations for DPA/CPA countermeasures on the GHASH function (BF_MUL and BF_ADD refer to field multiplication and field addition operations over $GF(2^{128})$, respectively).

	#BF_MUL	#BF_ADD
Oshida et al. [9]	$log(m+n+1)+2$	$m+n+5$
This Work (Alg. 5)	1	$m+n+3$

iteratively. By XORing the output of i-th BF_MUL where $1 \leq i \leq m+n$ with V, the output becomes dated as being always masked with $(M \oplus S)$. The final output of the GHASH function is XORed with U and then the authentication tag is returned. Note that by XORing U to the final output of the GHASH function, the random value M is removed from the output, which preserves the correctness property.

Table 5 compares the performance of the proposed masking method with Oshida *et al.*'s method. As mentioned previously in Sec. III-B, Oshida *et al.*'s method requires $\{log(m+n+1)+2\}$ BF_MULs and $(m+n+5)$ BF_ADDs for correctness property. Oshida *et al.* mentioned that $log(m+n+1)$ of BF_MULs can be saved by precomputing H^{m+n+1} prior to executing the GHASH function. However, since the number of input blocks in the GHASH function can vary, precomputing H^{m+n+1} does not always provide a computational advantage, in contrast to their assertion. However, the proposed masking method depicted in Alg. 5 requires additional overhead of $(m+n+3)$ BF_ADDs + (1) BF_MUL, which significantly reduces the number of the *BF* multiplications compared to Oshida *et al.*'s. Furthermore, since the cost for computing BF_ADD is much smaller than that for computing BF_MUL, the additional overhead from the proposed method is negligible compared with the original overhead for the GHASH computation itself. In the next section, we show the proposed countermeasure is resistant against DPA/CPA through experiments.

V. IMPLEMENTATION RESULT AND SECURITY ANALYSIS

In this section, firstly we compare the performance of the proposed secure *BF* multiplication method with those of the existing *BF* multiplication methods. Next, we compare the SCA security of the proposed GHASH function with that of the existing implementations of the GHASH function in terms of SPA/TA and DPA/CPA, respectively. For analyzing the performance and SCA Security, we have implemented AES-GCM software using our proposed BF multiplication method and DPA/CPA-resistant GHASH function on an SCARF SCA evaluation board [29] equipped with 8-bit ATmega128 microcontroller.

A. PERFORMANCE COMPARISON

Table 4 compares the cost and security of the proposed secure *BF* multiplication method with those of the existing methods. Oshida *et al.* used the Shift-and-Add method in order to compute the *BF* multiplications in the GHASH function [9]. However, they did not present the timing cost for computing the *BF* multiplications in their paper. Thus, we have implemented the Shift-and-Add method as described in Alg. 6 on an 8-bit ATmega128 microcontroller in C language and measured the performance. Since the Shift-and-Add method does not provide constant-timing, we have measured the average timing of 100 executions, and determined that it costs 31,971 *cc*. Even though the Shift-and-Add method does not require bit-reflection of the input operands and its structure is simple, it requires a number of bit-wise SHIFT and XOR instructions compared with the Comb-based methods, as described in Section II-C. Liu *et al.* presented the timing cost of their 128-bit *BF* multiplication method (named as Karatsuba MBC method) on an 8-bit AVR ATxmega128a1 microcontroller at Section 6 in their paper as 14,878 *cc*, including the cost for modular reduction [11]. Even though Liu *et al.*'s method provides constant-timing execution, it is vulnerable to SPA, as will be presented in the next section. We have implemented the proposed 128-bit *BF* multiplication method described in Section 2 on an 8-bit ATmega128 microcontroller. The cost of the proposed 128-bit *BF* multiplication method is 7,162 *cc*, which includes the overhead for bit-reflection and modular reduction (the cost for the 128-bit *BF* multiplication itself is 5,675 *cc*, as described in Table 2). The proposed method provides substantially improved performance by about 51.86% as compared with Liu *et al.*'s method while also providing security against TA and SPA. The codes for bit-reflection and modular reduction are implemented in C language. Thus, we expect that further improvement can be possible if they are implemented in AVR assembly language.

B. SIDE CHANNEL SECURITY ANALYSIS

In this section, we analyze the SCA security of the proposed *BF* multiplication method and the GHASH function. We also analyze the SCA security of the existing methods and compare them with that of our methods. For experiments, we have implemented three AES-GCM softwares: the first one using Liu *et al.*'s method, the second one using Oshida *et al.*'s method, and the final one using our methods on an SCARF

TABLE 4. Performance comparison of the 128-bit *BF* multiplication methods for the GHASH function on an 8-bit AVR microcontroller. Timing is measured with clock cycles (*cc*).

	Bit	Method	TA/SPA Security	Timing (cc)
Liu et al. [11]	128	Karatsuba + Masked Block Comb	TA only	14,878
Oshida et al. [9]	128	Shift-and-Add method	none	31,971
This Work	128	Karatsuba + Block-Comb with Dummy XOR and *ILA*	TA/SPA both	7,162

TABLE 5. Software configuration for experiments.

	Applied BF method in GHASH	GHASH function
Liu et al. [11]	Karatsuba + Masked Block Comb	Original GHASH
Oshida et al. [9]	Shift-and-Add method	Masked GHASH
This Work	Karatsuba + Block-Comb with Dummy XOR and *ILA*	Masked GHASH

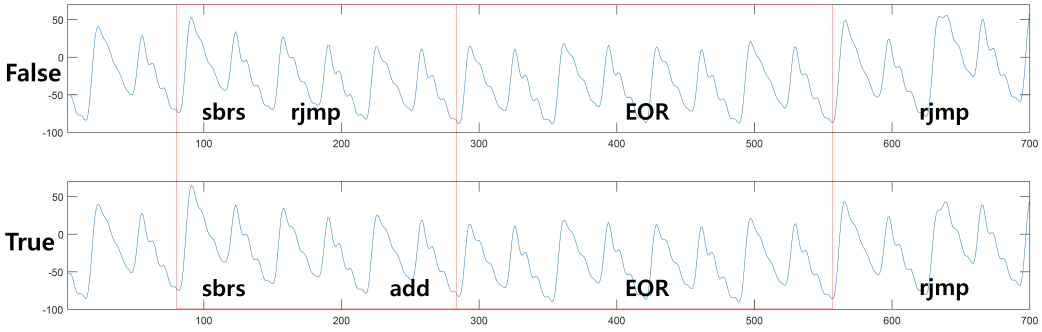


FIGURE 7. Power trace comparison between the false case and the true case.

SCA evaluation board [29] using 8-bit ATmega128 microcontroller, according to Table 5. The power consumption signal is captured at a 250MS/s sampling rate by LeCroy HDO6104A oscilloscope and then is low-pass filtered with 150MHz cut-off frequency. Actually, we have gathered power traces of overall AES-GCM execution and extracted subset of target points from the power traces. Note that since Liu *et al*. did not consider the DPA/CPA security of the GHASH function in their work, we use the original GHASH function without the DPA/CPA countermeasures for experimentation.

1) VALIDITY OF INSTRUCTION LEVEL ATOMICITY

As aforementioned in Section IV-A2, `SBRS` used in the *BC* method consumes different numbers of clock cycles depending on whether or not the tested bit is true. Thus, we use a dummy `ADD` instruction when the tested bit is true in order to eliminate the timing difference as compared to the false case. The reason why we make use of `ADD` rather than other AVR instructions is to make the True case indistinguishable from the False case with respect to SPA. Since the main goal of `SBRS` and `RJMP` is to update the *PC* (Program Counter) by adding the displacement value to the *PC* itself, the `ADD` instruction is the most appropriate for the dummy instruction. Fig. 7 compares the two power consumption traces between the False case and the True case when executing Alg. 3 in order to show the validity of the proposed ILA (Instruction Level Atomicity) against SPA and TA. The two cases shown in the figure consume the same number of clock cycles, and their power consumption patterns are indistinguishable from each other.

2) SPA SECURITY ANALYSIS

In order to analyze SPA security, we gather a power consumption trace of each *BF* multiplication method shown in Table 5 by using a LECROY HDO06104A oscilloscope with 5 GS/s.

Fig. 8 shows the power consumption trace of the Shift-and-Add method used in Oshida *et al*.'s GHASH function. In the figure, each black box includes the power consumption of each iteration of Alg. 6 in Appendix. The power consumption in a black box is divided into five distinct areas, and it can be found that when the multiplier bit is 1, the width of the first area is wider than that when the multiplier bit is 0. Thus, it can be found that the Shift-and-Add method used in Oshida *et al*.'s GHASH function is vulnerable to SPA.

In order to compare the proposed *BF* multiplication method with Liu *et al*.'s method, we make the two methods operate on the same operands. Fig. 9 shows the power consumption trace of Liu *et al*.'s *BF* multiplication method. In the figure, the power consumption of each iteration of the outer loop (from $l = 7$ to 0) in Alg. 1 is plotted in four consecutive black boxes. In other words, the power consumption in each black box is for computing each iteration of the inner loop (from $m = 3$ to 0) in Alg. 1. Note that since at the end of each iteration of the outer loop the accumulator C is left-shifted, the width of the fourth in successive four boxes is wider than that of the others. The power trace in the red box of the figure on the left side is magnified into the figure on the right side. As shown in the figure, when the multiplier bit is 1, there are several peaks at the right side of black box, otherwise there is no peak. Thus, the power consumption pattern when the multiplier bit is set can be clearly distinguishable from that when the multiplier bit is clear, resulting in vulnerability to SPA. This experimental result supports our security analysis on Liu *et al*.'s method described in Section III-B.

Fig. 10 shows the power consumption trace of the proposed *BF* multiplication method. From the figure, it can be found that the power consumption when the multiplier bit is set is

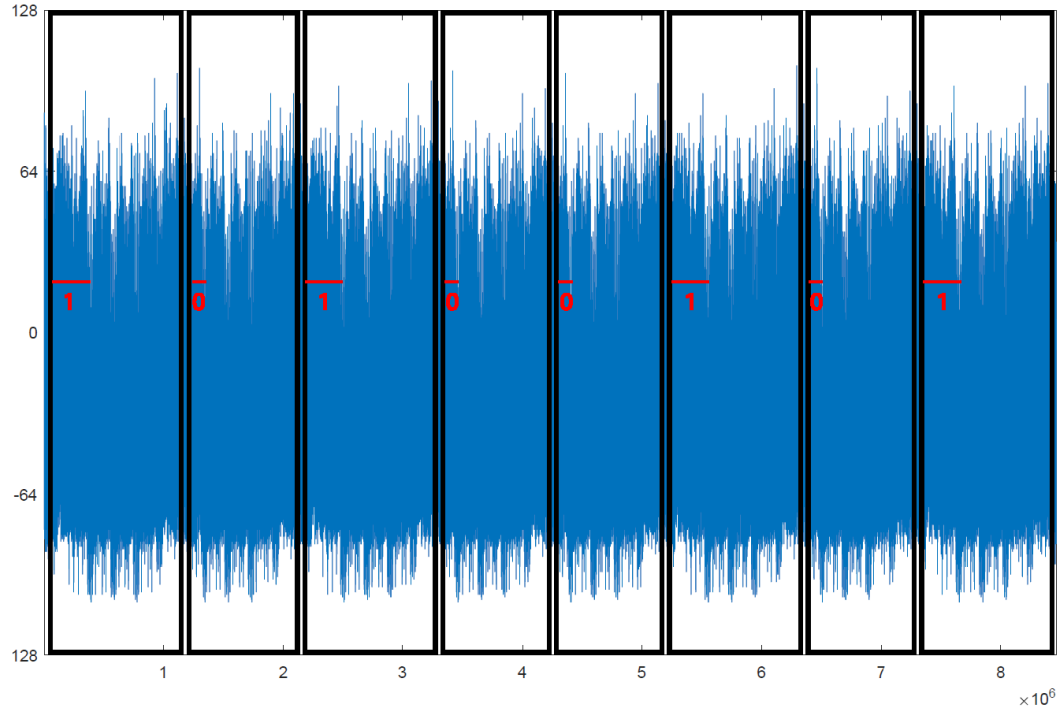


FIGURE 8. SPA on the shift-and-add method used in Oshida *et al.*'s method (The value 0 or 1 in the black box means the tested bit value of a multiplier).

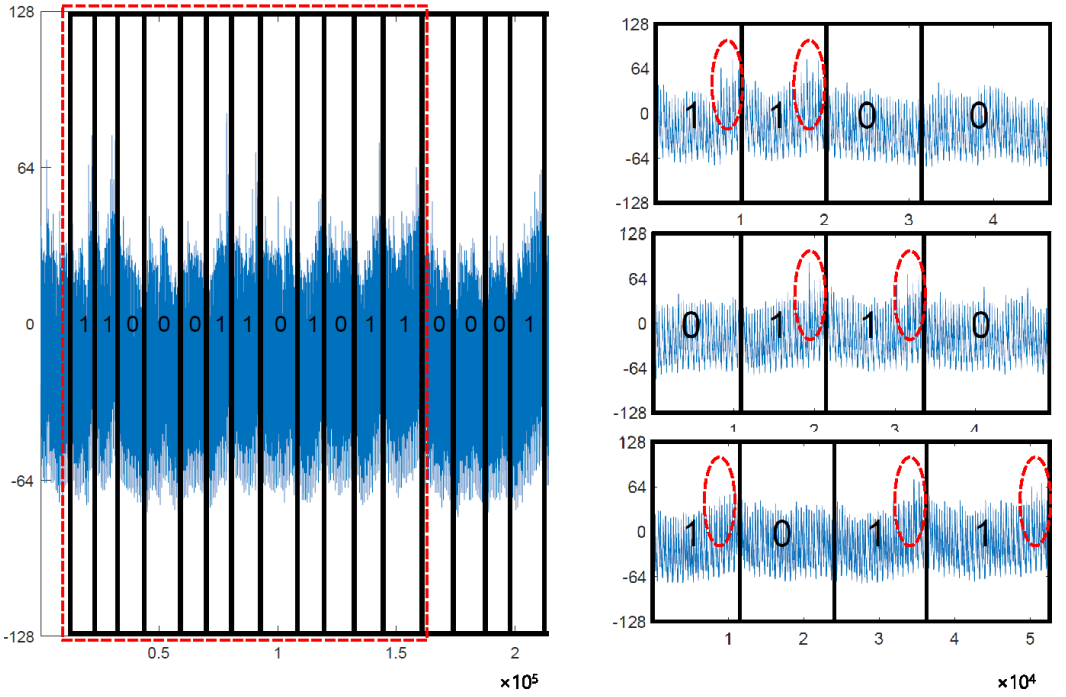


FIGURE 9. SPA on Liu *et al.*'s MBC method (The value 0 or 1 in each black box means the tested bit value of a multiplier. The power trace in the red box of the figure on the left side is magnified into the figure on the right side).

indistinguishable from that when the multiplier bit is clear, which ensures SPA-resistance. With the use of dummy registers and Instruction Level Atomicity (ILA), the proposed *BF* multiplication preserves the execution uniformity.

We have additionally checked that the proposed Instruction Level Atomicity (*ILA*) cannot be attacked with clustering algorithms such as K-Means [30], Spectral clustering [31], and so on. We have first classified the traces for assembly

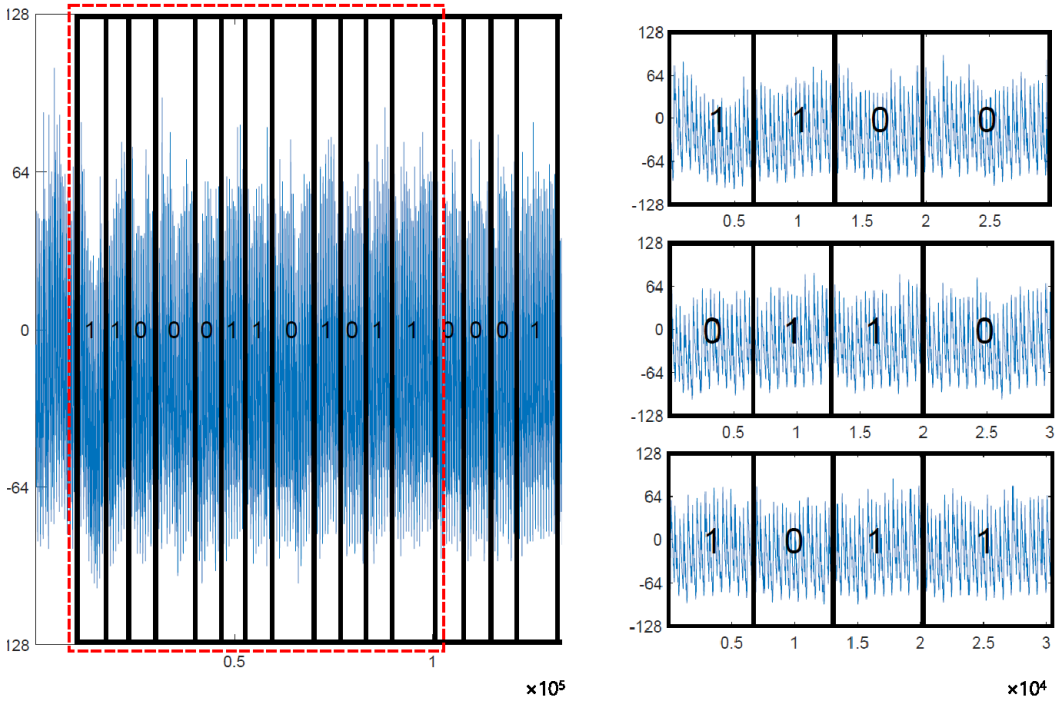


FIGURE 10. SPA on the proposed *BF* multiplication method (the value 0 or 1 in each black box means the tested bit value of the multiplier. The power trace in the red box of the figure on the left side is magnified into the figure on the right side).

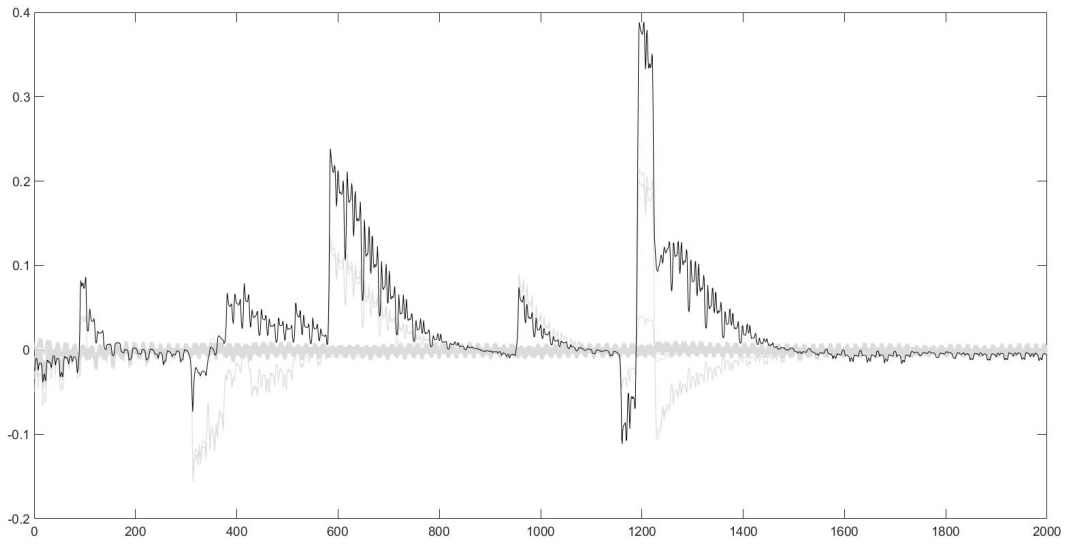


FIGURE 11. CPA on the original GHASH function using Liu *et al.*'s *BF* multiplication method (gray plots mean the correlations of wrong keys and the black plot means the correlation of a correct key).

codes in Fig. 5 into those for True case and False case assuming that the condition is known. With 5,000 traces for True case and 5,000 traces for False case, we have carried out the most widely used clustering algorithms: K-Means [30], Mean-Shift [32], DBSCAN [33], Spectral clustering [31], Birch [34], and Agglomerative clustering [35], [36]. In our experiments, two clustering algorithms K-Means and Spectral clustering have the highest success rates as 0.6109 and 0.6205, respectively. However, even though the probability that attackers are able to exactly guesses each bit of hash key H is 0.6205, the entropy of 128-bit H is $log_2 0.6205^{-128} = 88.128$ and it is still too high for attackers to find the hash key H in AES-GCM, which ensures the proposed method provides our claimed SCA security. Thus, our countermeasure can provide the practical security against various clustering techniques.

3) DPA/CPA SECURITY ANALYSIS

In order to analyze the DPA/CPA security, we gather 100,000 power consumption traces of each GHASH function in Table 3, the proposed GHASH function using the proposed *BF* multiplication method and the original GHASH

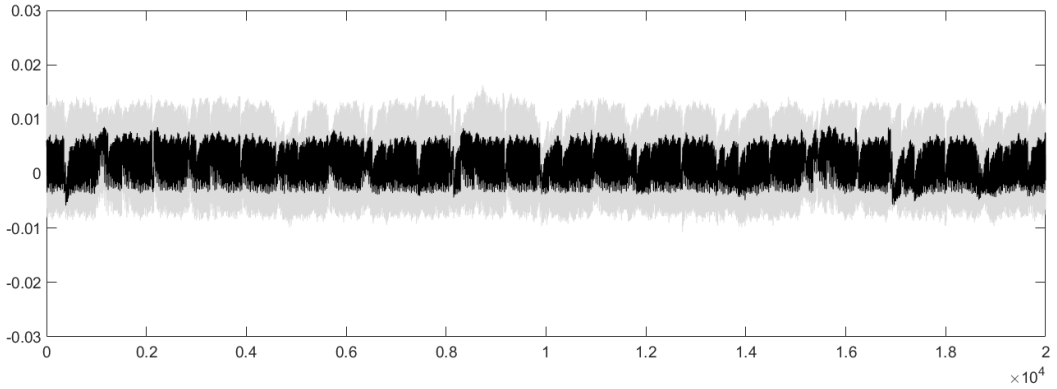


FIGURE 12. CPA on the proposed GHASH function using the proposed *BF* multiplication method (gray plots mean the correlations of wrong keys and the black plot means the correlation of a correct key).

function using Liu *et al.*'s *BF* method, by using LECROY HDO06104A oscilloscope with 500 MS/s. Since the DPA/CPA security of Oshida *et al.*'s method has been proven in their work [9], we omit conducting DPA/CPA on Oshida *et al.*'s masked GHASH function.

Fig. 11 shows correlations from conducting CPA on the original GHASH function using Liu *et al.*'s *BF* multiplication method. From the figure, it can be found that the only correct key, denoted by black has a higher correlation than any other correlations of wrong keys, denoted by gray. This result is apparent because Liu *et al.*'s work does not consider DPA/CPA security in the process of the GHASH function.

Fig. 12 shows the correlation result from executing CPA on the proposed GHASH function using the proposed *BF* multiplication method. As shown in the figure, the correlation of the correct key is indistinguishable from the other correlations of the wrong keys, which shows the security of the proposed method against DPA/CPA. From the experiments of SPA and DPA/CPA, comprehensive SCA security is required to achieve secure GCM implementation.

VI. CONCLUSION

In this paper, we propose a SCA-resistant GCM implementation for 8-bit AVR microcontroller environments. In recent years, researchers have realized the importance of SCA security on the GHASH function in the GCM mode and presented several attack methods. Even though several works have investigated developing SCA countermeasures for secure GHASH function, they failed to provide comprehensive SCA security against SPA/TA and DPA/CPA. In contrast to the existing methods, the proposed implementation provides comprehensive SCA and the proposed countermeasures are efficient in terms of computational cost. Since the binary field multiplications in $GF(2^{128})$ are the core operations in the GHASH function, we focus on developing a secure and efficient binary field multiplication method. For SPA/TA resistance, we introduce concepts of Dummy XOR with the garbage registers and instruction level atomicity (*ILA*), and present a secure binary field multiplication method using these concepts. The proposed field multiplication method provides substantially improved performance by about 51.86% as compared with the latest related works while providing TA/SPA resistance. Furthermore, we propose an efficient multiplicative masking method which can defend DPA/CPA during the process of the GHASH function. The proposed masked GHASH function also provides much improved performance compared with the latest related works. Through SPA/TA and DPA/CPA experiments, we prove the comprehensive SCA security of the proposed methods. In the future, we will extend the basic block size in the proposed Block-Comb method from 32-bit to 64-bit. By extending the block size, we expect that the number of partial products for computing a 128-bit binary field multiplication can be significantly reduced, which contributes to enhancing the overall performance of the GHASH function. Furthermore, we apply our method on 16-bit and 32-bit embedded microcontrollers such as MSP430 and ARM processors.

Algorithm 6 Shift-and-Xor Based *BF* Multiplication in $GF(2^{128})$

Require: 128-bit operands A and B where $A, A \in GF(2^{128})$
Ensure: Result $C{=}X \cdot B \bmod R$ where R is an irreducible polynomial as $f(z) = z^{128} + z^7 + z^2 + z + 1$.

```
C ← 0, V ← A
for i = 0 to 127 do
    if B_i == 1 then
        C ← C ⊕ V
    end if
    if V_127 == 0 then
        V ← rightshift(V)
    else
        V ← rightshift(V) ⊕ R
    end if
end for
(Return C)
```

APPENDIX

See Algorithm 6.

REFERENCES

[1] D. McGrew and J. Viega. (2005). *The Galois/Counter Mode of Operation (GCM)*. [Online]. Available: http://uca-giuzzi.unibs.it/corsi/Support/papers-cryptography/gcm-spec.pdf

[2] D. A. McGrew and J. Viega, "The security and performance of the galois/counter mode (GCM) of operation," in *Proc. Int. Conf. Cryptol., Prog. Cryptol.-INDOCRYPT*, in Lecture Notes in Computer Science, vol. 3348. New Delhi, India: Springer, 2004, pp. 343–355.

[3] M. J. Dworkin, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (Gcm) and GmAC*, document NIST Special 800-38D, 2007.

[4] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. Annu. Int. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, 1999, pp. 388–397.

[5] S. Belaïd, P. Fouque, and B. Gérard, "Side-channel analysis of multiplications in GF(2^{128})," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.* Kaoshiung, Taiwan : Springer, 2014, pp. 306–325.

[6] S. Belaïd, J.-S. Coron, P.-A. Fouque, B. Gérard, J.-G. Kammerer, and E. Prouff, "Improved side-channel analysis of finite-field multiplication," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Saint-Malo, France: Springer, 2015, pp. 395–415.

[7] P. Pessl and S. Mangard, "Enhancing side-channel analysis of binary-field multiplication with bit reliability," in *Proc. Cryptographers Track RSA Conf.* San Francisco, CA, USA: Springer, 2016, pp. 255–270.

[8] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Lausanne, Switzerland: Springer, 2009, pp. 1–17.

[9] H. Oshida, R. Ueno, N. Homma, and T. Aoki, "On masked galois-field multiplication for authenticated encryption resistant to side channel analysis," in *Proc. Int. Workshop Constructive Side Channel Anal. Secure Design*. Singapore: Springer, 2018, pp. 44–57.

[10] H. Seo, C.-N. Chen, Z. Liu, Y. Nogami, T. Park, J. Choi, and H. Kim, "Secure binary field multiplication," in *Proc. 16th Int. Workshop Inf. Secur. Appl.* Jeju Island, Korea: Springer, 2015, pp. 161–173.

[11] Z. Liu, H. Seo, C.-N. Chen, Y. Nogami, T. Park, J. Choi, and H. Kim, "Secure GCM implementation on AVR," *Discrete Appl. Math.*, vol. 241, pp. 58–66, May 2018.

[12] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, Cambridge, MA, USA, Springer, 2004, pp. 16–29.

[13] A. Joux, "Authentication failures in NIST version of GCM," Gaithersburg, MD, USA: NIST, Tech. Rep., 2006. [Online]. Available: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf

[14] Atmel. *AVR Instruction Set Manual*. Accessed: Jul. 31, 2019. [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

[15] J. López and R. Dahab, "High-speed software multiplication in F_{2m}," in *Proc. Int. Conf. Cryptol.*, in Lecture Notes in Computer Science, vol. 1977. New Delhi, India: Springer, 2000, pp. 203–212.

[16] S. C. Seo, D.-G. Han, H. C. Kim, and S. Hong, "TinyECCK: Efficient elliptic curve cryptography implementation over GF(2^m) on 8-bit Micaz mote," *IEICE Trans. Inf. Syst.*, vol. E91-D, no. 5, pp. 1338–1347, May 2008.

[17] D. F. Aranha, R. Dahab, J. López, and L. B. Oliveira, "Efficient implementation of elliptic curve cryptography in wireless sensors," *Adv. Math. Commun.*, vol. 4, no. 2, pp. 169–187, May 2010.

[18] L. B. Oliveira, D. F. Aranha, C. P. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab, "TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks," *Comput. Commun.*, vol. 34, no. 3, pp. 485–493, Mar. 2011.

[19] M. Shirase, Y. Miyazaki, T. Takagi, D.-G. Han, and D. Choi, "Efficient implementation of pairing-based cryptography on a sensor node," *IEICE Trans. Inf. Syst.*, vol. E92, no. 5, pp. 909–917, 2009.

[20] H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim, "Binary and prime field multiplication for public key cryptography on embedded microprocessors," *Secur. Commun. Netw.*, vol. 7, no. 4, pp. 774–787, Apr. 2014.

[21] H. Seo, Z. Liu, J. Choi, and H. Kim, "Karatsuba–block-comb technique for elliptic curve cryptography over binary fields," *Secur. Commun. Netw.*, vol. 8, no. 17, pp. 3121–3130, Nov. 2015.

[22] S. C. Seo and H. Seo, "Highly efficient implementation of NIST-compliant Koblitz curve for 8-bit AVR-based sensor nodes," *IEEE Access*, vol. 6, pp. 67637–67652, 2018.

[23] D. C. Hankerson, A. Menezes, and S. A. Vanstone, *Guide to Elliptic Curve Cryptography*. Montreal, QC, Canada: Springer, 2006.

[24] C. P. L. Gouvêa and J. López, "High speed implementation of authenticated encryption for the MSP430X microcontroller," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.* Santiago, Chile: Springer, 2012, pp. 288–304.

[25] C.-N. Chen, "Memory address side-channel analysis on exponentiation," in *Proc. Int. Conf. Inf. Secur. Cryptol. (ICISC)*, in Lecture Notes in Computer Science, vol. 8949. Seoul, South Korea: Springer, 2014, pp. 421–432.

[26] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil, "Horizontal correlation analysis on exponentiation," in *Proc. Int. Conf. Inf. Commun. Secur.* Barcelona, Spain: Springer, 2010, pp. 46–61.

[27] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Sov. Phys. Doklady*, vol. 7, pp. 595–596, Jan. 1963.

[28] P. L. Montgomery, "Five, six, and seven-term Karatsuba-like formulae," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 362–369, Mar. 2005.

[29] Y. Choi, D. Cho, and J. Ryou, "Implementing side channel analysis evaluation boards of KLA-SCARF system," *J. Korea Inst. Inf. Secur. Cryptol.*, vol. 24, no. 1, pp. 229–240, 2014.

[30] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, 1967, pp. 281–297.

[31] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, Aug. 2000.

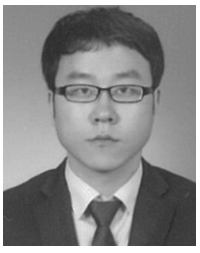
[32] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 17, no. 8, pp. 790–799, Aug. 1995.

[33] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discovery Data Mining*, Aug. 1996, pp. 226–231.

[34] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 1996, pp. 103–114.

[35] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 18, no. 7, pp. 790–799, 1995.

[36] D. Defays, "An efficient algorithm for a complete link method," *Comput. J.*, vol. 20, no. 4, pp. 364–366, 1977.



SEOG CHUNG SEO received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, and the M.S. degree in information and communications from the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2005 and 2007, respectively, and the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. He was a Research Staff Member of the Samsung Advanced Institute of Technology (SAIT) and the Samsung DMC Research and Development Center, from 2011 to 2014. He was a Senior Research Member of the Affiliated Institute of ETRI, South Korea, from 2014 to 2018. He is currently an Assistant Professor with Kookmin University, South Korea. His research interests include public-key cryptography and its efficient implementations on various IT devices, cryptographic module validation programs, network security, and data authentication algorithms.

HEESEOK KIM received the B.S. degree in mathematics from Yonsei University, Seoul, South Korea, in 2006, and the M.S. and Ph.D. degrees in engineering and information security from Korea University, Seoul, South Korea, in 2008 and 2011, respectively. He was a Postdoctoral Researcher with the University of Bristol, U.K., from 2011 to 2012. From 2013 to 2016, he was a Senior Researcher with the Korea Institute of Science and Technology Information (KISTI). Since 2016, he has been with Korea University. His research interests include side-channel attacks, cryptography, and network security.