Highly Efficient SCA-Resistant Binary Field Multiplication on 8-Bit AVR Microcontrollers

Seog Chung Seo 1,*, and Donggeun Kwon 2

1 Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul 02707, Korea
2 Graduate School of Information Security and Institute of Cyber Security & Privacy (ICSP), Korea University, Seoul 02841, Korea; donggeunkwon@korea.ac.kr
* Correspondence: scseo@kookmin.ac.kr

Received: 25 March 2020; Accepted: 16 April 2020; Published: 19 April 2020

Abstract: Binary field (BF) multiplication is a basic and important operation for widely used crypto algorithms such as the GHASH function of GCM (Galois/Counter Mode) mode and NIST-compliant binary Elliptic Curve Cryptosystems (ECCs). Recently, Seo et al. proposed a novel SCA-resistant binary field multiplication method in the context of GHASH optimization in AES GCM mode on 8-bit AVR microcontrollers (MCUs). They proposed a concept of Dummy XOR operation with a kind of garbage registers and a concept of instruction level atomicity (ILA) for resistance against Timing Analysis (TA) and Simple Power Analysis (SPA) and used a Karatsuba Block-Comb multiplication approach for efficiency. Even though their method achieved a large performance improvement compared with previous works, it still has room for improvement on the 8-bit AVR platform. In this paper, we propose a more improved binary field multiplication method on 8-bit AVR MCUs. Our method basically adopts a Dummy XOR technique using a set of garbage registers for TA and SPA security; however, we save the number of used garbage registers from eight to one by using the fact that the number of used garbage registers does not affect TA and SPA security. In addition, we apply a multiplier encoding approach so as to decrease the number of required registers when accessing the multiplier, which enables the use of extended block size in the Karatsuba Block-Comb multiplication technique. Actually, the proposed technique extends the block size from four to eight and the proposed binary field multiplication method can compute a 128-bit BF multiplication with only 3816 clock cycles (cc) (resp. 3490 cc) with (resp. without) the multiplier encoding process, which is almost a 32.8% (resp. 38.5%) improvement compared with 5675 cc of the best previous work. We apply the proposed technique to the GHASH function of the GCM mode with several additional optimization techniques. The proposed GHASH implementation provides improved performance by over 42% compared with the previous best result. The concept of the proposed BF method can be extended to other MCUs, including 16-bit MSP430 MCUs and 32-bit ARM MCUs.

Keywords: authenticated encryption; binary field arithmetic; GHASH function; Galois/Counter Mode of Operation; Simple Power Analysis; Timing Analysis; Binary Elliptic Curve Cryptosystem; 8-bit AVR; ATmega128

1. Introduction

Binary field (BF) multiplication is an important and the most time-consuming arithmetic operation in several widely used cryptographic algorithms, including the Galois/Counter mode (GCM) operation and NIST-compliant binary ECC (Elliptic Curve Cryptosystems). For example, the central computation part of the GHASH function in GCM is consecutive 128-bit BF multiplications, and BF multiplications also occupy almost 80% of the running time of binary Elliptic Curve Cryptosystems, such as ECDH (Elliptic Curve Diffie–Hellman), ECDSA (Elliptic Curve Digital Signature Algorithm), ECIES (Elliptic
Curve Integrated Encryption Scheme), and ECMQV (Elliptic Curve Menezes–Qu–Vanstone) (ECDH and ECMQV are key agreement schemes based on Diffie–Hellman and MQV, respectively. ECIES is known as the Elliptic Curve Augmented Encryption Scheme, which can be used as a key transport mechanism. ECDSA is a digital signature algorithm based on the Digital Signature Algorithm. Details of the aforementioned Elliptic Curve-based algorithms can be found in [1]).

However, the representatives of 8 bit, 16 bit, and 32 bit embedded microcontrollers, including AVR, MSP430, and ARM Cortex-M, do not have a dedicated multiplier for the binary field multiplication. For this reason, the software implementation of the multiplication over a binary field has been considered to be a more challenging task than that of the multiplication over a prime field. Furthermore, 8-bit AVR and 16-bit MSP430 MCUs are very resource-limited regarding memory and computation capacity. Thus, designing an optimal _BF_ multiplication method on such MCUs is a challenging task. Nonetheless, until now, several efficient _BF_ multiplication methods have been proposed on 8-bit AVR MCUs and they are classified into two main categories: Lookup Table (LUT)-based approaches [2–5] and Block-Comb (BC)-based approaches [6–10].

Among several optimization techniques [2–9], the Karatsuba Block-Comb (KBC) multiplication method was shown that it could provide better performance compared with the LUT-based approach [2–5] when calculating a multiplication over _GF_(2^{163}). From the works in [6–8], the maximum block size is widely known as seven words on 8-bit AVR microcontrollers. However, recently Seo et al. have proposed an enhanced Karatsuba Block-Comb multiplication technique and shown how to extend the block size from seven to eight with their novel multiplier encoding technique [9]. By expanding the block size, they could decrease the number of partial products required to calculate a multiplication over _GF_(2^{233}) in the context of a NIST-compliant Binary K-233 Curve. Even though Seo et al.’s multiplication method over a binary field provides the best performance among existing _BF_ multiplication methods, it does not provide resistance against Timing Analysis (TA) and Simple Power Analysis (SPA).

There have been some works regarding the SCA (side channel analysis) security of _BF_ multiplication [10,11]. Liu et al. showed that LUT-based multiplication methods are vulnerable to horizontal Correlation Power Analysis (CPA) [12] and proposed a masked Block-Comb multiplication method, which does not use any LUT while providing TA and SPA resistance by eliminating a conditional loop in its execution [11]. Their method is used for the GCM mode of operation and it computes a 128-bit _BF_ multiplication in 14,445 cc. Most recently, Seo et al. have shown that the masked Block-Comb multiplication method from Liu et al. actually does not provide SPA security and proposed a novel secure Block-Comb (SBC) multiplication method with a new concept of Dummy XOR operations using a set of garbage registers and a concept of instruction level atomicity for both TA and SPA security [10]. Seo et al.’s method can compute a multiplication over _GF_(2^{128}) in 5675 cc, and its TA and SPA security are proven to be secure. Although Seo et al.’s method is efficient and resistant against TA and SPA, the performance of their method still needs to be improved considering low performance of the 8-bit AVR MCUs. Actually, the block size of Seo et al.’s method is just four, which is much lower than the known maximum block size of the enhanced Karatsuba Block-Comb multiplication method on 8-bit AVR devices (Seo et al.’s method’s basic multiplication unit is 32-bit wise. Thus, nine partial products are required for calculating _BF_ multiplication over _GF_(2^{128}) with the Karatsuba technique). Thus, so as to improve the performance, it is necessary to expand the block size of the secure Block-Comb multiplication technique.

This paper presents an enhanced secure _BF_ multiplication method on 8-bit AVR MCUs broadly used for RFIDs, smart cards, embedded controllers, and wireless sensor nodes. Our method is a kind of Block-Comb multiplication method and adopts a technique of Dummy XOR operations using a set of garbage registers and the concept of instruction level atomicity (ILA). Through experiments, we find out that the number of used garbage registers does not affect SPA security. Therefore, it is possible to decrease the number of necessary garbage registers from eight to one and makes use of the saved registers to improve the performance of the _BF_ multiplication. In order to make the most of
the available registers on 8-bit AVR MCUs, we apply a multiplier encoding approach that can further curtail the number of registers to access the multiplier during multiplication from $s$ to just one (where $s$ is the block size of Block-Comb multiplication method). Thus, we can expand the block size of the secure Block-Comb multiplication method from four to eight the same as the known maximum block size on 8-bit AVR MCUs. As a result, we can decrease the number of partial multiplications from nine to three when calculating a field multiplication over $GF(2^{128})$, which results in a large performance improvement.

1.1. Research Contributions

The following are our summarized contributions.

1. **Presenting an enhanced secure Block-Comb multiplication method on 8-bit AVR MCUs**
   We present an enhanced secure Block-Comb multiplication method on 8-bit AVR MCUs. Through experiments with SPA traces analysis and security analysis using clustering algorithms, we show that the number of used garbage registers does not affect SPA security. With this fact, we configure that our method makes use of a single garbage register, which saves seven registers. In order to further extend the block size of our Block-Comb method, we apply a multiplier encoding technique for optimizing the usage of the registers on 8-bit AVR MCUs. As a result, we extend the block size of the secure Block-Comb multiplication method from four (32-bit wise) to eight (64-bit wise), identical to the maximum block size on AVR MCUs, which significantly decreases the number of partial multiplications from nine to three when calculating a $BF$ multiplication over $GF(2^{128})$. The proposed method can be a building block for binary field multiplication in the GHASH function of GCM and NIST-compliant binary ECC.

2. **Implementing the proposed secure Block-Comb multiplication method on an ATmega128 MCU**
   By implementing on 8-bit ATmega128, we show that our proposed multiplication technique consumes much less running time. In our method, the basic multiplication unit is 64-bit wise and we apply an enhanced Karatsuba technique when calculating a 128-bit wise $BF$ multiplication. The proposed method takes 3816 $cc$, including a multiplier encoding process, when computing a 128-bit $BF$ multiplication, almost 32.8% faster than that of Seo et al.’s (Consumes 5675 $cc$), while providing TA/SPA security. Without the multiplier encoding process, the proposed takes 3490 $cc$, which is almost a 38.5% improvement of Seo et al.’s method.

3. **Application to GHASH function of GCM mode**
   We show how to apply the proposed $BF$ method to the GHASH function of GCM. Even though our method requires a multiplier encoding process before executing $BF$ multiplication, this process can be omitted in the context of the GHASH function. In other words, in the 128-bit binary field multiplication of the GHASH function, one of the two inputs is fixed as a Hash key. Thus, we can apply multiplier encoding to the hash key and store the encoded hash key in memory before starting the GHASH function, and reuse it during the GHASH function process. In the GHASH function, the inputs and output of $BF$ multiplication need to be bit-reflected, which requires three bit-reflection operations per $BF$ multiplication. We propose a technique that can decrease the count of bit-reflection from three to one. We present an enhanced GHASH function implementation on ATmega128, and it provides an improved performance by over 42% compared to the previous best result.

1.2. Comparison to the Previous Work

Even though the previous work first introduced the concept of a dummy XOR operation with garbage registers and instruction level atomicity (ILA) [10], its performance was still low. This is because the work in [10] utilized the block size of four and the naive Karatsuba technique. Furthermore, the process of the GHASH function was not optimized in the work of [10]. Whereas the basic concept of our current work is similar to previous work, our current proposed method applies several
optimization techniques crucial for improving the performance of a binary field multiplication and
GHASH function in the GCM of operation. Firstly, we have shown that the number of garbage registers
does not affect SPA security by conducting in-depth experiments. By reducing the number of garbage
registers from eight to one and applying a multiplier encoding technique, the proposed method could
achieve block size eight, which is the well-known maximum block size in the Block-Comb method on
8-bit AVR MCUs. Furthermore, by further optimizing the execution of the Karatsuba technique and the
process of the GHASH function, our method could achieve significant performance improvement. As a
result, the currently proposed method has achieved performance improvement over 42% compared
with the previous best result while providing the same level of security.

The remainder of our paper is composed as follows. Section 2 introduces characteristics of
8-bit AVR microprocessors, and a multiplication over a binary field. Section 3 describes existing BF
multiplication algorithms on 8-bit AVR MCUs. Section 4 presents the proposed secure and efficient
multiplication method over a binary field on 8-bit AVR MCUs. Both the proposed methods are
analyzed with respect to performance and security in Section 4. Section 5 describes the proposed
GHASH function implementation with the proposed multiplication method on 8-bit AVR MCUs. Section 6 describes a concluding remark with future works.

2. Related Works

This section introduces the characteristics of 8-bit AVR MCUs regarding the number of registers,
memory size, and AVR instruction set. Then, we describe the basics of the BF multiplication methods.
There are two main categories of BF multiplication approaches: LookUp Table-based (LUT-based)
approaches and Block-Comb-based (BC-based) approaches. A detailed description of the existing
multiplication methods over a binary field on 8-bit AVR MCUs will be given in Section 3.

Eight-Bit AVR Microcontrollers and Notations

Currently, devices using 8-bit AVR are broadly used for diverse applications, like RFIDs,
smartcards, embedded controllers, wireless sensor nodes, and so on. Typically, 8-bit AVR MCUs,
including our target platform ATmega128, contain 32 general-purpose registers ($R_{31}, \ldots, R_1, R_0$).
Among 32 registers, six registers are utilized as memory address pointers. Each set of ($R_{26}, R_{27}$),
($R_{28}, R_{29}$), and ($R_{30}, R_{31}$) are aliased as $X$, $Y$, and $Z$ pointer registers, respectively [13]. Typically, AVR
MCUs have not only individual memory spaces but also buses for data and program instructions in a
simple single-issued pipeline manner, since their architecture is based on the Harvard architecture.
There are 133 instructions in total, and typically, each instruction executes in constant latency.
For instance, logical/arithmetic instructions (e.g., ROR (rotate right through carry), LSL (logical shift
left), EOR (bit-wise XOR), ADD (arithmetic add), and so forth) are executed within a single clock cycle,
while instructions related to memory accesses (e.g., ST (store from register to memory), LD (load from
memory to register), and so on) consume two clock cycles [13]. In the case of conditional branch
instructions, their clock cycles depend on whether the tested condition is true or not. For instance,
in the case of SBRS (skip next instruction if bit in register is set), if the condition is true, it takes up
two or three cycles depending on the skipped instruction’s word size. Otherwise, it consumes one
cycle. The memory and computation capabilities of 8-bit AVR MCUs are limited. For example, an 8-bit
ATmega128 MCU has 4 Kbytes of RAM and 128 Kbytes of ROM memory, and its running clock speed
is 7.3728 MHz. Contrary to the state-of-the-art ARM MCUs and Intel CPUs providing a carryless
multiplier and generic binary field hardware multiplier, AVR MCUs still do not embed the dedicated
hardware multiplier.

All through our paper, the following notations are used. The general purpose registers are
represented as $R$. $R_i$ means to the $i$-th general-purpose register in which $0 \leq i \leq 31$. The operators $\gg$
and $\ll$ mean logical right shifts, logical left shifts, and XOR operation. $A[i]$ refers to the $A$’s $i$-th byte
and it consists of 8 bits like $(a_{8i+7}, \ldots, a_{8i})$. Finally, $A[n, \ldots, m]$ corresponds to the bytes from $A[m]$ to
$A[n]$, respectively.
Multiplication over Binary Field

Binary Field (BF) multiplication is a core operation of several cryptographic algorithms, such as the GHASH function of GCM and NIST-compliant binary elliptic curve operations. For example, in the GHASH function of GCM, BF multiplications are executed with input operands as associated data blocks or ciphertext blocks, and a secret constant hash key \( H \). In the case of binary ECC, scalar multiplication is the most performance-critical part of the entire ECC-based protocols and its almost 80% running time comes from BF multiplications. Thus, the performance of BF multiplication needs to be optimized as much as possible.

\( BF \) multiplication computes \( A \cdot B \) where \( A = \sum_{i=0}^{m-1} a_i z^i \), \( B = \sum_{i=0}^{m-1} b_i z^i \in GF(2^m) \) where \( m \) is the degree of the underlying binary extension field. In the above notation, each multiplicand and multiplier are represented as \( A \) and \( B \), respectively. The result of the multiplication can be represented as \( C = \sum_{i=0}^{m-1} A \cdot b_i z^i \). The most basic algorithm for a multiplication over a binary field is the Shift-and-Add method. It scans the multiplier from LSB (Least Significant Bit, the 0-th bit) to MSB (Most Significant Bit, the \((m-1)\)-th bit). At every bit, multiplicand \( A \) is shifted in the left direction like \( A \cdot z \), and if the bit of multiplier \( B \) is 1, the accumulator is XORed with \( A \cdot z \) (Namely, if \( b_i \), the multiplier \( B \)'i-th bit, is 1, the accumulator is XORed with \( A \cdot z^i \)). The Comb multiplication algorithm, the basic algorithm for both LUT-based multiplication algorithms, and Block-Comb-based multiplication algorithms enhance the performance of binary field multiplication. Actually, Comb multiplication algorithms make use of the fact that \( A \cdot z^{w_i+k} \) can be easily attained by adding \( j \) zero words to the right side of the vector representation of \( A \cdot z^{i} \), once \( A \cdot z^{i} \) has been computed for some \( k \in [0, W−1] \) (\( W \) is eight in case of 8-bit AVR). Therefore, it can decrease the count of shift operations as compared to the Shift-and-Add method. Two categories of Comb methods exist: the RTL version and the LtR version. While the RTL version of the Comb method scans a multiplier from LSB to MSB, the LtR version of the Comb method operates the other way round \([2,14]\).

3. Multiplication Methods over \( GF(2^m) \) on 8-bit AVR MCUs

Until now, many studies have been conducted for optimizing \( BF \) multiplication’s performance on 8-bit AVR platforms \([2–8]\). They can be categorized into two main approaches: LookUp Table-based (LUT-based) approaches \([2–5]\) and Block-Comb-based (BC-based) approaches \([6–9]\). Table 1 summarizes the existing result results, and the details will be explained in the following Sections 3.1 and 3.2. Because the count of accessible registers is constrained on 8-bit AVR, many memory accesses take place. Namely, among 32 general-purpose registers, only 26 are accessible for calculating a \( BF \) multiplication, without six registers for a memory address pointer. For instance, at least a set of 64 registers are necessary for maintaining the total part of a multiplier, a multiplicand, and a result of multiplication. However, due to the limited number of available registers, only certain parts of the operands can be kept in the registers. Thus, this limitation generates a huge number of redundant memory accesses. Therefore, on 8-bit AVR, the major goal of existing researches on binary field multiplication methods is minimizing redundant memory accesses by optimizing the use of the available registers.

| Technique | Application | Fields | Timing (cc) | SCA Resistance |
|-----------|-------------|--------|-------------|----------------|
| Seo et al. [3] | LookUp-Table (4-bit wise) | ECC \( GF(2^{2163}) \) | 19,670 | SPA, TA |
| Aranha et al. [4] | LookUp-Table (4-bit wise) | ECC \( GF(2^{2163}) \) | 4508 | SPA, TA |
| Shirase et al. [6] | Block-Comb | ECC \( GF(2^{232}) \) | 9511 | none |
| Seo et al. [7] | Unbalanced Block-Comb | ECC \( GF(2^{163}) \) | 4346 | none |
| Seo et al. [8] | Karatsuba Block-Comb | ECC \( GF(2^{163}) \) | 3274 | none |
| Seo et al. [8] | Constant Karatsuba Block-Comb | ECC \( GF(2^{163}) \) | 5005 | TA |
| Seo et al. [9] | Enhanced Karatsuba Block-Comb | ECC \( GF(2^{2233}) \) | 6896 | none |
| Ziu et al. [11,15] | Masked Block-Comb | GCM \( GF(2^{128}) \) | 14,445 | TA |
| Seo et al. [10] | Block-Comb with Dummy XOR and \( ILA \) | GCM \( GF(2^{128}) \) | 5675 | SPA, TA |
3.1. Look-Up Table-Based Methods

So, as to enhance the performance of field multiplications of the GHASH function in GCM, firstly McGrew et al. presented a table-based method using different sizes considering the trade-off between computational speed and memory consumption [16,17] in their GCM implementation. They used different sized tables: a version of 256 bytes, a version of 4 Kbytes, a version of 8 Kbytes, and a version of 64 Kbytes and measured the performance on a 32-bit Motorola G4 device. Although their table-based methods are efficient regarding computational speed, memory consumption is too huge to be utilized on 8-bit AVR MCUs. Therefore, researchers usually have taken advantage of López et al.’s Look-Up Table multiplication method [3–5], originally aimed for field multiplication of binary elliptic curves operation [2,14] when implementing the GCM algorithm on resource-limited embedded devices, including AVR and MSP430.

López et al.’s LUT-based technique is an extended version of LtR Comb technique (it is called the wLtR Comb technique) [2,14]. The wLtR Comb technique calculates a multiplication by w-bit wise rather than single-bit wise at the cost of building a precomputation table. Thus, it can reduce the count of bit operations, like bit XOR and shift operations [2,4,14,18]. At the beginning of the multiplication, it computes all possible results of \( A \cdot u(z) \) about all polynomials \( u(z) \) of degree at most \( w - 1 \) and stores them in a kind of precomputation table. Then, in the actual multiplication process, multiplier \( B \) is scanned by \( w \)-bit at a time from the left (MSB) to right (LSB) direction, and the corresponding value from the precomputation table is chosen. Namely, the corresponding value from the table is XORed with the intermediate value in the accumulator without actual computation. On 8-bit AVR MCUs, it is widely believed that 4-bit is the most favorable width \( w \) for this wLtR Comb technique. Therefore, it makes use of \( 16 \times m \)-bit of RAM memory for maintaining the precomputation table consisting of sixteen multiplication results from \( 0 \cdot A \) to \( (z^3 + z^2 + z + 1) \cdot A \). This LUT-based method and its variants have been broadly implemented on 8-bit AVR devices [3–5]. For instance, 163-bit binary field multiplication was implemented by NesC language on an ATmega128 MCU in Seo et al.’s work [3], and it was upgraded by integrating two iterations of the main loop into one, decreasing the count of redundant memory accesses. Seo et al. got 19,670 cc as the timing result for a field multiplication over \( GF(2^{163}) \), which was a 21.1% improvement. In [4,5], Aranha et al. proposed a concept of a rotating register technique in the wLtR Comb method, and it could greatly decrease the count of redundant memory accesses necessary for executing a multiplication method. Their method was implemented in AVR Assembly language and they reported 4508 cc, 8314 cc, and 11,727 cc for calculating a multiplication over \( GF(2^{163}) \), \( GF(2^{233}) \), and \( GF(2^{271}) \), respectively. LUT-based multiplication methods give not only good performance but also are resistant against both TA and SPA. However, they are vulnerable to side channel attacks, which uses information about the memory address [11,12,19] owing to the large number of resulted memory accesses. In [11,15], Liu et al. successfully analyzed the wLtR Comb multiplication technique with a sort of horizontal correlation analysis [12]. Namely, they could get the indices used for accessing LUT by using the correlation between power consumption traces from building up the Lookup table and referencing the LUT element during the process of a multiplication.

3.2. Block-Comb Based Multiplication Methods

As an alternative to binary field multiplication using LUT, the Block-Comb (BC) multiplication method was originally proposed for efficient field multiplication of \( \eta_T \) pairing over \( GF(2^{239}) \) on an ATmega128 MCU [6]. In the BC multiplication method, both the multiplier and the multiplicand of multiplication are partitioned into \( s \)-byte blocks. Then, partial products of generated blocks are calculated by a column-wise fashion. Each of the partial products is calculated with the LtR Comb method for performance efficiency. Namely, in the BC multiplication method, the set of accessible registers are partitioned into three parts; \( s \) registers, \( s \) registers, and \( 2s + 1 \). These partitioned three parts of registers are used for a multiplicand, a multiplier, and the result of the partial multiplication, respectively. Because the intermediate results are kept in the set of working registers of \( 2s + 1 \), the results of partial multiplications positioned in the identical column can be updated directly into
the registers without accessing memory, which decreases the count of redundant memory accesses. In [6], Shirase et al. drew a conclusion that six is the optimum block size s based on the fact that 

\[(4s + 1) < 26\] on 8-bit AVR MCUs. Shirase et al.’s BC multiplication method calculates a multiplication over \(GF(2^{259})\) in 9511 clock cycles (cc).

Seo et al. [7] proposed the Unbalanced Block-Comb multiplication method (UBC), which can expand the block size from 6 to 7 for a multiplication over \(GF(2^{163})\). They exploited the fact that the tested bits of a multiplier become unnecessary during a partial product computation process. In other words, they recycled the register used to keep the multiplier in order to hold the multiplicand’s the most significant byte. Consequently, the expanded block size decreases the count of partial multiplication from sixteen to nine when calculating a field multiplication over \(GF(2^{163})\). Note that block size 7 (resp. block size 6) partitions a 163-bit field element into 3 blocks (resp. 4 blocks). Seo et al.’s UBC could calculate a 163-bit binary multiplication within 4546 cc. Then, Seo et al. [8] presented the so-called Karatsuba Block-Comb multiplication method (KBC), an integration of the Karatsuba technique and the Block-Comb multiplication approach, which decreases the count of partial multiplications from nine to six at the cost of several low cost field additions when calculating a multiplication over \(GF(2^{163})\). KBC could accomplish 3274 cc for a multiplication over \(GF(2^{163})\). They also proposed a constant version of their Karatsuba Block Comb multiplication technique. Although it accomplished resistance against a timing attack, it can still be attacked by a simple power analysis. In 2018, Seo et al. [9] proposed an enhanced version of the Karatsuba Block-Comb (EKBC) multiplication technique by applying a new multiplier encoding method, which can greatly decrease the count of registers necessary for keeping the multiplier. In addition, they showed that with their proposed technique, the maximum block size of the Block-Comb multiplication method could be 8 on 8-bit AVR MCUs. As a result, they accomplished a new timing record for NIST-compliant K-233 elliptic curve scalar multiplication. Until now, EKBC is regarded as the fastest multiplication method over a binary field on 8-bit AVR MCUs.

### 3.3. Secure Block-Comb Multiplication Methods

Since Block-Comb multiplication methods do not utilize any Lookup Table, they have resistance against a sort of horizontal correlation analysis [11,15], which was used for analyzing LUT-based methods. However, they are vulnerable to TA and SPA because they contain a conditional branch. Algorithm A1 in Appendix A shows a simple 56-bit wise Block-Comb multiplication method. Steps 11–15 are executed only when the l-th bit of the multiplier is 1, which is the source of TA and SPA vulnerability.

In 2018, Liu et al. [11,15] presented a masked version of the Block-Comb (MBC) technique for secure GCM implementation on 8-bit AVR MCUs. Algorithm A1 is the masked Block-Comb method and it basically operates on 32-bit wise (Algorithm 1). The algorithm replaces the conditional branch with masked XOR operations for SPA and TA resistance. For instance, if the tested bit (BIT) at step 3 is 0, then zero values \((B[m]\&0x00)\) are XORed with the accumulator \(C\) through steps 5–7. Otherwise, normal values \((B[m]\&0xFF)\) are XORed with the accumulator \(C\). They combined the MBC with the Karatsuba technique for computing a 128-bit BF multiplication. It is reported that Liu et al.’s 128-bit BF multiplication took 14,878 cc. Even though MBC achieves TA security, it still has the vulnerability of SPA in contrast to Liu et al.’s assertion. In other words, XORing with zero value \((B[m]\&0x00)\) when the tested bit is 0 has a different power consumption pattern compared with XORing with original value \((B[m]\&0xFF)\) when the tested bit is 1. Seo et al. showed that Liu et al.’s MBC is still vulnerable to SPA in their recent work [10].

Most recently, Seo et al. proposed a novel secure Block-Comb method, which has resistance against TA and SPA (in addition, their method is resistant against a sort of horizontal CPA because it does not use any Lookup Table) for a secure GCM implementation on 8-bit AVR MCUs [10]. Similar to MBC, seo et al.’s method has made use of a 32-bit wise Block-Comb multiplication method. For making a 32-bit wise Block-Comb multiplication method secure against SPA, they introduced the concept of Dummy XOR operations with a set of garbage registers. In other words, the count of registers for the accumulator \(C\) is doubled from 8 registers to 16 registers \((R_{15}, \ldots, R_0)\). Thus, their method
makes use of 25 registers in total \((R_{15}, \ldots, R_0)\) for accumulator \(C\), \((R_{20}, \ldots, R_{16})\) for multiplicand \(A\), and \((R_{24}, \ldots, R_{21})\) for multiplier \(B\), which is acceptable in 8-bit AVR MCUs having 32 registers. Among \((R_{15}, \ldots, R_0)\) registers, the set of \((R_7, \ldots, R_0)\) plays the role of the garbage registers, and the set of \((R_{15}, \ldots, R_8)\) maintains the real intermediate result of the multiplication. With the Dummy XOR operations with garbage registers, the multiplicand is XORed at a different position relying on the value of the tested bit. For instance, if the tested bit is 0, the registers \((R_{20}, \ldots, R_{16})\) containing the multiplicand are XORed with the garbage registers \((R_7, \ldots, R_0)\). Otherwise, the same registers are XORed with the part of real accumulator \((R_{15}, \ldots, R_8)\). Since the registers keeping real multiplicand values are XORed with the accumulator registers in both cases, the power consumption patterns for both cases are not distinguishable each other with respect to SPA. In order to implement this concept of Dummy XOR as being secure against TA, they introduced the concept of instruction level atomicity (ILA). On 8-bit AVR MCUs, typically the branch instructions consume different clock cycles relying on whether the tested condition is true or not. For instance, if the condition is false (resp. true), it usually takes 1 clock cycle (resp. 2 clock cycles). They identified that the main role of the branch instruction is to increment the program counter (PC) depending on whether the condition is true or not, and used a dummy ADD instruction to fill the timing difference. Even though their method uses SBRS branch instruction, the timing difference is hidden by the dummy ADD instruction. Seo et al. use their 32-bit wise Block-Comb multiplication method for calculating 128-bit \(BF\) multiplication. For efficiency, they applied a two level Karatsuba technique, which consists of nine 32-bit partial products and each partial product is computed by their proposed Block-Comb multiplication method. They reported the timing cost of 128-bit \(BF\) multiplication as 5675 cc.

Algorithm 1: 32-bit wise Masked Block-Comb \([11,15]\).

| Require: 32-bit multiplier \(A\) and 32-bit multiplicand \(B\) |
| Ensure: 64-bit result \(C = A \cdot B\) |
| 1: for \(k = 7\) to \(0\) do |
| 2: for \(n = 3\) to \(0\) do |
| 3: \(BIT \leftarrow A[n] & (1 \ll k)\) |
| 4: \(MASK, T0 \leftarrow (0 - BIT)\) |
| 5: for \(m = 3\) to \(0\) do |
| 6: \(C[m + n] \leftarrow C[m + n] \oplus (B[m] & MASK)\) |
| 7: end for |
| 8: end for |
| 9: \(C \leftarrow C \ll 1\) |
| 10: end for |
| 11: (Return C) |

Table 1 shows the existing implementation of \(BF\) multiplication on 8-bit AVR MCUs.

4. Proposed Binary Field Multiplication

In this section, we describe the proposed \(BF\) multiplication method, which is not only efficient but also secure against TA and SPA. With several optimization techniques, we present a secure Block-Comb method using block size 8 known as the maximum block size on 8-bit AVR MCUs.

4.1. Enhanced Secure Block-Comb Method

Seo et al.’s utilized \(n\) garbage registers equal to the number of real accumulator registers (namely, eight registers were used as the garbage register set). Our method makes use of a single garbage register rather than using \(n\) garbage register. The security analysis of using a single garbage register is described in Section 4.4. The saved registers can be used for extending the block size of the Block-Comb method. If the Block-Comb method with a garbage register uses block size \(s\), the number of total registers is
(4s + 2): 1, s, s + 1, 2s are for the garbage register, the multiplier, the multiplicand, and the accumulator. Since on 8-bit AVR MCUs 26 registers are available except for address registers, the block size s can be 6 (48-bit). Since Seo et al. utilized the 32-bit secure Block-Comb technique for calculating 128-bit BF multiplication, 9 partial products were required (actually, they integrate their secure Block-Comb method into the Karatsuba technique. Thus, 16 partial products are reduced into 9 partial products). With the 48-bit wise Block-Comb method, 128-bit operands are divided into three terms. Thus, the 128-bit BF multiplication can be computed with 6 partial products by integration with the Karatsuba technique. Even though by using a garbage register the block size of the secure Block-Comb method has been extended from 4 to 6, it still does not reach the maximum block size 8, which was presented from the work of the non-constant Enhanced Karatsuba Block-Comb method [9].

In order to expand the block size of the Block-Comb multiplication method from 6 to 8, it is required to reserve more registers. However, since on 8-bit AVR MCUs the available registers are only 26 except for address registers, one of register sets for maintaining each of multiplier, multiplicand, or accumulator needs to be reduced. Common Block-Comb methods load s bytes of the multiplier into s registers (at Step 4~7 of Algorithm A1 in Appendix A) and sequentially access l-th bit of s registers where l is from 0 to 7 (at Step 11 of Algorithm A1 in Appendix A). Figure 1 shows the multiplier accessing pattern of Algorithm A1 in Appendix A. Since the bits being accessed are distributed in s registers, they require s registers for accessing the multiplier. Thus, we encode the multiplier so that the bits being accessed are in one register [9]. In other words, by rearranging l-th bit of s registers into one register, steps 10~16 of Algorithm A1, the main inner loop of the Block-Comb method, requires only one register at a time. Algorithm 2 depicts a 128-bit wise multiplier encoding process. The algorithm makes use of AVR bit handling instructions as LSR (logical shift right), ROR (rotate right through carry). Figure 2 shows the encoded multiplier from bit reordering-based multiplier encoding process when the multiplier is 128-bit, which can be used for BF multiplication in the GHASH function of GCM. The encoding process operates on 64-bit wise so that i-th encoded byte EB[i] contains 0-th bit of original multiplier B[0] ∼ B[7]. With the application of multiplier encoding, our method loads l-th bit of original multiplier’s s bytes into one register. Thus, our method requires (3s+3) registers for computing a Block-Comb method (namely, 1, 1, s + 1, and 2s registers are for the multiplier, the garbage register, the multiplicand, and real accumulator, respectively). Even though s value is 7, satisfying 3s + 3 ≤ 26, we can extend the block size from 7 to 8 by using exploiting the address registers (r31, . . ., r26). Actually, address registers can be used as arithmetic registers. In other words, the memory address for the final result is needed only when storing the final multiplication result in the accumulators into the memory at the end of the BF multiplication. Thus, at the beginning of the multiplication, the address for the final result can be stored at the stack memory with PUSH instruction and then be restored with POP instruction when storing the final result into the memory. This technique requires only 8 cc because 2 PUSH and 2 POP instructions are used for storing and restoring the 16-bit address value. Therefore, we make the secure Block-Comb method use block size 8, the same as the maximum block size reported from the work of the enhanced Karatsuba Block-Comb method [9].

Figure 1. Multiplier accessing pattern in common Block-Comb methods.
Algorithm 2: 128-bit multiplier encoding technique on 8-bit AVR MCUs.

**Require:** 128-bit Multiplier $B$ over $GF(2^{128})$.

**Ensure:** Encoded multiplier $EB$.

1: for $j = 0$ to 1 do

2: Load $R_{2j}, \ldots, R_0 \leftarrow B[8j + 7, \ldots, 8j]$

3: for $i = 0$ to 7 do

4: for $n = 0$ to 7 do

5: $LSR R_n$

6: $ROR R_i$

7: end for

8: Store $R_8$ at $EB[8j + i]$

9: end for

10: end for

11: return $(EB)$

---

**Figure 2.** The result of the 128-bit multiplier encoding process. **Left**-side: original input multiplier, **Right**-side: encoded multiplier.

**Figure 3** shows the register configuration of the proposed Block-Comb method.

Algorithm 3 depicts the proposed secure Block-Comb method using the block size 8 (64-bit wise). We assume that the multiplier $B$ is converted into $EB$ with Algorithm 2 and then $EB$ is used as the input of multiplier in Algorithm 3. In the algorithm, $(R_{24}, \ldots, R_{16}), (R_{25}, R_{26}),$ and $(R_{15}, \ldots, R_0)$ hold the multiplicand $A$, the encoded multiplier $EB$, the garbage register, and the intermediate result, respectively. Note that Algorithm 3 makes use of a single register for keeping target bits of the multiplier because each byte of $EB$ has each bit column of the original multiplier’s consecutive eight bytes.
Algorithm 3: Proposed 64-bit Block-Comb technique with the bit-reordering technique where \((R_{15}, \ldots, R_0)\), \((R_{24}, \ldots, R_{16})\), and \((R_{25})\) are used for the accumulator, multiplicand, and encoded multiplier.

**Require:** 64-bit multiplicand \(A\) and 64-bit encoded multiplier \(EB\)

**Ensure:** 128-bit result \(C(128\text{-bit})=A \cdot B\)

1. \(R_{27} \leftarrow 0x0A\) // Set a displacement value in a register for \(ILA\)
2. \(\text{for } i = 0 \text{ to } 15 \text{ do}\)
3. \(R_i \leftarrow 0\) // Initialize accumulator \(C\)
4. \(\text{end for}\)
5. \(\text{for } i = 0 \text{ to } 7 \text{ do}\)
6. \(R_{16+i} \leftarrow A[i]\) // Load multiplicand \(A\)
7. \(\text{end for}\)
8. \(R_{24} \leftarrow 0\)
9. // Processing 0–6-th bit of the multiplier
10. \(\text{for } i = 0 \text{ to } 6 \text{ do}\)
11. \(R_{25} \leftarrow EB[i]\) // Load encoded multiplier \(EB\)
12. \(\text{for } n = 0 \text{ to } 7 \text{ do}\)
13. \(\text{if the } n\text{-th bit of } R_{25} = 1 \text{ then}\)
14. \(R_{26} \leftarrow R_{26} + R_{27}\) // Dummy ADD operation for \(ILA\)
15. \(\text{for } k = 0 \text{ to } 8 \text{ do}\)
16. \(R_{n+k} \leftarrow R_{n+k} \oplus R_{16+k}\)
17. \(\text{end for}\)
18. \(\text{else}\)
19. // Dummy XOR operation with a garbage register
20. \(\text{for } k = 0 \text{ to } 8 \text{ do}\)
21. \(R_{26} \leftarrow R_{26} \oplus R_{16+k}\)
22. \(\text{end for}\)
23. \(\text{end if}\)
24. \(\text{end for}\)
25. \((R_{24}, \ldots, R_{16}) \leftarrow (R_{24}, \ldots, R_{16}) \ll 1\)
26. \(\text{end for}\)
27. // Processing the final bit of the multiplier
28. \(R_{25} \leftarrow EB[7]\) // Load encoded multiplier \(EB\)
29. \(\text{for } n = 0 \text{ to } 7 \text{ do}\)
30. \(\text{if the } n\text{-th bit of } R_{25} = 1 \text{ then}\)
31. \(R_{26} \leftarrow R_{26} + R_{27}\) // Dummy ADD operation for \(ILA\)
32. \(\text{for } k = 0 \text{ to } 8 \text{ do}\)
33. \(R_{n+k} \leftarrow R_{n+k} \oplus R_{16+k}\)
34. \(\text{end for}\)
35. \(\text{else}\)
36. // Dummy XOR operation with a garbage register
37. \(\text{for } k = 0 \text{ to } 8 \text{ do}\)
38. \(R_{26} \leftarrow R_{26} \oplus R_{16+k}\)
39. \(\text{end for}\)
40. \(\text{end if}\)
41. \(\text{end for}\)
42. \((\text{Return } C = (R_{15}, \ldots, R_0))\)
4.2. Proposed Karatsuba Technique

With the proposed Block-Comb multiplication method using a block size of 8, 128-bit BF multiplication can be computed with 4 partial products, where each partial product is computed with Algorithm 3. To decrease the count of partial products, we combine our Block-Comb method with the enhanced Karatsuba technique [20]. Actually, even though the enhanced Karatsuba technique was originally proposed for prime field multiplication on 8-bit AVR MCUs, we modify it for our proposed Block-Comb method. By applying the enhanced Karatsuba technique rather than classic Karatsuba technique, s XOR instructions can be saved where s is the number of words for operands and 8 in Algorithm 4. Algorithm 4 depicts the proposed 1-level Karatsuba secure Block-Comb method, which computes 3 partial products with Algorithm 3. In the algorithm, \( H, H_L, \) and \( M \) mean low term, high term, and middle term in the Karatsuba multiplication, respectively (thus, each of them is 128-bit). Lower case \( L \) and \( H \) mean the lower part and higher part of each term (thus, each of them is 64-bit).

Algorithm 4: 128-bit wise Karatsuba Block Comb method.

Require: \( 128\)-bit wise operands \( A \) and \( B \).
Ensure: Result \( C(128\text{-bit})=A \cdot B \).
1. Encode \( B[7 \ldots 0] \) into \( EB[7 \ldots 0] \) with Algorithm 2
2. Encode \( B[15 \ldots 8] \) into \( EB[15 \ldots 8] \) with Algorithm 2
3. \( L_H \vert\vert L_L \leftarrow A[7 \ldots 0] \times_{64}\text{-bit} EB[7 \ldots 0] \) with Algorithm 3
4. \( H_H \vert\vert H_L \leftarrow A[15 \ldots 8] \times_{64}\text{-bit} EB[15 \ldots 8] \) with Algorithm 3
5. \( T \leftarrow L_H \oplus H_L \)
6. \( L_H \leftarrow T \oplus L_L \)
7. \( H_L \leftarrow T \oplus H_H \)
8. \( M_H \vert\vert M_L \leftarrow (A[15 \ldots 8] \oplus A[7 \ldots 0]) \times_{64}\text{-bit} (EB[15 \ldots 8] \oplus EB[7 \ldots 0]) \) with Algorithm 3
9. \( L_H \leftarrow L_H \oplus M_L \)
10. \( H_L \leftarrow H_L \oplus M_H \)
11. \( C \leftarrow (H_H \vert\vert H_L \vert\vert L_H \vert\vert L_L) \)
12. (Return \( C \))

4.3. Implementation Results on an 8-Bit ATmega128 MCU and Comparison to Previous Work

We have implemented our methods on a target board containing ATmega128 MCU. Table 2 shows the timing results of the proposed methods and compare them with those of Seo et al.’s method [10]. For efficiency, the proposed methods are developed in AVR assembly language. A 64-bit wise proposed secure Block-Comb method requires 1213 \( cc \) (resp. 1050 \( cc \)) with (resp. without) multiplier encoding process, and these timing results are an 8.8% (resp. 21.1%) improvement compared with Seo et al.’s 64-bit wise multiplication method. In the case of 128-bit multiplication, the proposed method with (resp. without) multiplier encoding requires 3816 \( cc \) (resp. 3490 \( cc \)), which are improvements of 32.8% and 38.5% compared with Seo et al.’s 128-bit wise multiplication method. The improvement ratio in 128-bit multiplication has been increased compared with that in 64-bit multiplication. This is because we implement 128-bit multiplication by the enhanced Karatsuba technique with assembly language while Seo et al.’s 128-bit multiplication method is implemented with C and assembly language. Furthermore, the partial product at step 8 in Algorithm 4 does not require a multiplier encoding process because the encoded multiplier can be obtained by XORing the precomputed multipliers as \( (EB[15 \ldots 8] \oplus EB[7 \ldots 0]) \).
Table 2. Timing analysis and comparison for the proposed SCA-resistant 64-bit, and 128-bit wise BF multiplication (the cost includes the overhead for function calls, such as POP and PUSH instructions). ME means multiplier encoding.

| Bit     | Method                        | Timing (cc) | Improvement | Language |
|---------|-------------------------------|-------------|-------------|----------|
| 64-bit  | Seo et al.’s Karatsuba Block-Comb (Level 1) | 1330        | -           | ASM      |
| 128-bit | Seo et al.’s Karatsuba Block-Comb (Level 2) | 5675        | -           | ASM+C    |
| 64-bit  | Proposed Block-Comb with ME   | 1213        | 8.8%        | ASM      |
| 64-bit  | Proposed Block-Comb without ME| 1050        | 21.1%       | ASM      |
| 128-bit | Proposed Karatsuba Block-Comb (Level 1) with ME | 3816        | 32.8%       | ASM      |
| 128-bit | Proposed Karatsuba Block-Comb (Level 1) without ME | 3490        | 38.5%       | ASM      |

4.4. Security Impact Analysis According to the Number of Registers Used

We find out that the number of garbage registers does not affect SPA security on the target 8-bit AVR MCU, and we reduce the number of garbage registers from eight to just one. We have conducted SPA security analysis for proving the security impact of reducing the number of garbage registers. We have utilized the KLA-SCARF evaluation board having an 8-bit ATmega128 MCU and gather power traces with a LeCroy HDO06104A oscilloscope with a sampling rate 500 MS/s. Figure 4 compares two power consumption traces between the False and the True cases when using a single garbage register rather than \( n \) garbage registers for SPA and TA security. Both cases shown in the figure take the same number of clock cycles, and the two power consumption patterns are not distinguishable from each other.

![Figure 4. Comparison of power consumption patterns between the False case and the True case when using a single garbage register.](image)

We have additionally investigated that the proposed BF multiplication method cannot be analyzed with popular clustering algorithms like K-Means and Spectral clustering algorithms. In other words, firstly we have classified the traces for True case (XOR operations with the real accumulator) and False case (XOR operations with a garbage register), assuming that the condition is already known. By using 5000 traces for True case and 5000 traces for False case, we have conducted the most popular clustering algorithms: K-Means and Spectral clustering. In our experiments, each of the K-Means and Spectral clustering algorithms has success rates of 0.6385 and 0.6023, respectively. With a success rate of 0.6385, the entropy is 82.85 (\( \log_2 0.6385^{-128} \)), which is still a high security level with respect to SPA security.

5. Application to GCM Mode’s GHASH Function Implementation

We have applied the proposed BF multiplication method to the GHASH function of GCM as a case study. Since the GHASH function of GCM requires several 128-bit BF multiplications, the BF needs to be implemented in a secure and efficient way. Even though Seo et al. proposed an efficient and secure BF multiplication method for secure GHASH function of GCM of operation [10], the performance of their method needs to be improved for efficient GHASH function. Our method is more efficient than...
Seo et al.'s method by 38.5% while providing TA and SPA security. We also apply our method to the
GHASH function and suggest some optimization methods specific to GCM.

Since the hash key $H$ is fixed in the GHASH function, it can be encoded at the beginning of the GHASH
function, and the encoded $H$ can be used for $BF$ multiplications in the GHASH function without encoding
$H$ every time. Thus, we can save the overhead for encoding the multiplier of the $BF$ multiplications in
the GHASH function. In GCM standard, the bits in the state are reflected. In other words, the leftmost
bit is considered as the 0-th bit and the rightmost bit is considered as the 127-th bit, while general crypto
algorithms typically utilize the opposite notation. Therefore, two inputs of $BF$ multiplication need to be
bit-reflected and the output is also required to be bit-reflected. We apply a table-based bit-reflection method
(it requires 256-byte table). However, compared with Seo et al.'s method using three bit-reflections for two
inputs and an output, our implementation requires only one bit-reflection.

Figure 5 describes the process of GHASH function of GCM. The inputs of $BF$ multiplication
(each of $(A_1, \ldots, A_m, C_1, \ldots, C_n)$ and $H$) and the output $(V_1, \ldots, V_{m+n})$ need to be bit-reflected.
Our implementation encodes the hash key $H$ at the beginning of the GHASH function and stores
it in bit-reflected form, which removes the need for bit-reflection of $H$ at each $BF$ multiplication.
Our implementation combines the bit-reflection of $BF$ multiplication output and the bit-reflection of
one input $(A_1, \ldots, A_m, C_1, \ldots, C_n)$. Since the output of previous $BF$ multiplication is XORed with the
one input of the next $BF$ multiplication, we can combine two bit-reflection into one. In other words,
$\text{BitReflect}(O) \oplus \text{BitReflect}(I) = \text{BitReflect}(I \oplus O)$ where $\text{BitReflect}$ is a bit-reflection function
and, $O$ is the output of the previous $BF$ multiplication, and $I$ is the input of the next $BF$ multiplication.
Thus, in our implementation, only one bit-reflection is required at each $BF$ multiplication, which can
improve GHASH function's performance further.

We have measured the performance of the proposed GHASH implementation on an 8-bit ATmega128
MCU for each of 16 byte, 64 byte, and 256 byte messages and compare the results with those of Seo et al.'s
work. Even though Seo et al. used the C version of the 128-bit reduction method in their $BF$ multiplication
method [10], we have implemented a fast reduction method similar to the fast reduction algorithm in [14]
in assembly language, and the running time of the reduction is 350 cc. Note that including the final one
additional $BF$ multiplication for computing $V_{m+n+1}$ in Figure 5, 2, 5, and 17 128-bit $BF$ multiplications are
required for 16 byte, 64 byte, and 256 byte messages, respectively. Table 3 compares the performance of
our GHASH implementation and the previous work. Actually, since Seo et al. did not provide timing
results for 64 byte and 256 byte messages in their paper [10], and we have implemented their method and
measured the timings for 64 byte and 256 byte messages. Our proposed implementation provides improved
performance by over 42% compared to the previous best result from [9].

Table 3. Timing costs for GHASH function and comparison to previous work (timings are measured
by clock cycles (cc)).

|            | 16 Bytes | 64 Bytes | 256 Bytes |
|------------|----------|----------|-----------|
| Seo et al.'s implementation [10] | 13,816   | 34,156   | 115,516   |
| This work  | 7834     | 19,594   | 66,634    |
| Improvement ratio | 43.30%   | 42.6%    | 42.3%     |
6. Conclusions

In this paper, we have proposed a highly efficient SCA-resistant binary field multiplication method and applied it to the GHASH function of GCM on 8-bit AVR MCUs. The proposed \( BF \) multiplication method is efficient and secure against TA and SPA. Our method has adopted a concept of the Dummy XOR technique using a set of garbage registers and reduced the number of garbage registers from \( n \) to one by investigating the security impact of using only one garbage register. Furthermore, with a novel multiplier encoding, our method has achieved block size 8, which is known as the largest block size in the Block-Comb multiplication method on 8-bit AVR MCUs. As a result, our \( BF \) method presents an improved performance by 32.8\% (resp. 38.5\%) with the multiplier encoding (resp. without multiplier encoding) compared with the previous best work. The proposed \( BF \) multiplication method can be used as underlying \( BF \) multiplication in the GHASH function of GCM and NIST-compliant binary ECC arithmetic. As a case study, we have also presented enhanced GHASH function implementation with the proposed \( BF \) multiplication method and additional optimization techniques, which can improve the performance by over 42\% compared with the previous best GHASH function implementation.

As future works, we will apply the concept of the proposed method on 16-bit and 32-bit embedded MCUs, such as MSP430 and ARM processors. Furthermore, we will apply our proposed \( BF \) multiplication method to NIST-compliant Binary Elliptic Curve Cryptosystems (ECCs).

Author Contributions: S.C.S.: conceptualization, methodology, software implementation, paper writing, and project management, D.K.: SCA analysis and result verification. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A10558494).

Acknowledgments: We appreciate the anonymous reviewers for their valuable comments.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Algorithm A1: 56-bit wise Block-Comb multiplication method on ATmega128 microprocessor.

\( (R_{13}, \ldots, R_0), (R_{21}, \ldots, R_{14}), \) and \( (R_{28}, \ldots, R_{22}) \) are reserved for holding the accumulator \( C \), multiplicand \( A \), and multiplier \( B \), respectively.

\begin{itemize}
    \item [Require:] 56-bit multiplicand \( A \) and 56-bit multiplier \( B \).
    \item [Ensure:] 112-bit result \( C = A \cdot B \).
\end{itemize}

\begin{algorithmic}
\State \textbf{for} \( i = 0 \) to 13 \textbf{do}
\State Load \( R_i \leftarrow 0 \)
\State \textbf{end for}
\State \textbf{for} \( i = 0 \) to 6 \textbf{do}
\State Load \( R_{14+i} \leftarrow A[i] \)
\State Load \( R_{22+i} \leftarrow B[i] \)
\State \textbf{end for}
\State \( R_{21} \leftarrow 0 \)
\State \textbf{for} \( l = 0 \) to 7 \textbf{do}
\State \textbf{for} \( k = 0 \) to 6 \textbf{do}
\State if the \( l \)-th bit of \( R_{22+k} \)\( ==1 \) then
\State \textbf{for} \( m = 0 \) to 7 \textbf{do}
\State \( R_{k+m} \leftarrow R_{k+m} \oplus R_{14+m} \)
\State \textbf{end for}
\State \textbf{end if}
\State \textbf{end for}
\State if \( i \neq 7 \) then
\State \( (R_{21}, \ldots, R_{14}) \leftarrow (R_{21}, \ldots, R_{14}) \ll 1 \)
\State \textbf{end if}
\State \textbf{end for}
\State Return \( C \)
\end{algorithmic}
References

1. NIST. Cryptographic Algorithm Validation Program. Available online: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program (accessed on 18 April 2020).
2. López, J.; Dahab, R. High-Speed Software Multiplication in \( F_{2^m} \). In Proceedings of the International Conference on Cryptology in India, Calcutta, India, 10–13 December 2000; pp. 203–212.
3. Seo, S.C.; Han, D.G.; Kim, H.C.; Hong, S. TinyECCK: Efficient elliptic curve cryptography implementation over GF(\(2^m\)) on 8-bit Micaz mote. IEICE Trans. Inf. Syst. 2008, 91, 1338–1347. [CrossRef]
4. Aranha, D.F.; Dahab, R.; López, J.; Oliveira, L.B. Efficient implementation of elliptic curve cryptography in wireless sensors. Adv. Math. Comm. 2010, 4, 169–187. [CrossRef]
5. Oliveira, L.B.; Aranha, D.F.; Gouvêa, C.P.; Scott, M.; Câmara, D.F.; López, J.; Dahab, R. TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. Comput. Commun. 2011, 34, 485–493. [CrossRef]
6. Shirase, M.; Miyazaki, Y.; Takagi, T.; Han, D.G.; Choi, D. Efficient implementation of pairing-based cryptography on a sensor node. IEICE Trans. Inf. Syst. 2009, 92, 909–917. [CrossRef]
7. Seo, H.; Lee, Y.; Kim, H.; Park, T.; Kim, H. Binary and prime field multiplication for public key cryptography on embedded microprocessors. Secur. Commun. Netw. 2014, 7, 774–787. [CrossRef]
8. Seo, H.; Liu, Z.; Choi, J.; Kim, H. Karatsuba–Block-Comb technique for elliptic curve cryptography over binary fields. Secur. Commun. Netw. 2015, 8, 3121–3130. [CrossRef]
9. Seo, S.C.; Seo, H. Highly Efficient Implementation of NIST-Compliant Koblitz Curve for 8-bit AVR-Based Sensor Nodes. IEEE Access 2018, 6, 67637–67652. [CrossRef]
10. Seo, S.C.; Kim, H. SCA-Resistant GCM Implementation on 8-Bit AVR Microcontrollers. IEEE Access 2019, 7, 103961–103978. [CrossRef]
11. Liu, Z.; Seo, H.; Chen, C.; Nogami, Y.; Park, T.; Choi, J.; Kim, H. Secure GCM implementation on AVR. Discret. Appl. Math. 2018, 241, 58–66. [CrossRef]
12. Clavier, C.; Feix, B.; Gagnerot, G.; Roussellet, M.; Verneuil, V. Horizontal Correlation Analysis on Exponentiation. In Proceedings of the Information and Communications Security—12th International Conference (ICICS 2010), Barcelona, Spain, 15–17 December 2010; pp. 46–61.
13. Atmel. AVR Instruction Set Manual. Available online: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf (accessed on 18 April 2020).
14. Hankerson, D.; Menezes, A.J.; Vanstone, S. Guide to Elliptic Curve Cryptography; Springer Science: New York, NY, USA, 2004.
15. Seo, H.; Chen, C.; Liu, Z.; Nogami, Y.; Park, T.; Choi, J.; Kim, H. Secure Binary Field Multiplication. In Proceedings of the Information Security Applications—16th International Workshop, (WISA 2015), Jeju Island, Korea, 20–22 August 2015; pp. 161–173.
16. McGrew, D.; Viega, J. The Galois/Counter Mode of Operation. GCM. 2005. Available online: http://lucagiuzzi.unibs.it/corsi/Support/papers-cryptography/gcm-spec.pdf (accessed on 18 April 2020).
17. McGrew, D.A.; Viega, J. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Proceedings of the Progress in Cryptology—INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, 20–22 December 2004; pp. 343–355.
18. Gouvêa, C.P.L.; López, J. High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller. In Proceedings of the Progress in Cryptology—LATINCRYPT 2012—2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, 7–10 October 2012; pp. 288–304.
19. Chen, C.N. Memory address side-channel analysis on exponentiation. In Proceedings of the International Conference on Information Security and Cryptology; Springer: Cham, Switzerland, 2014; pp. 421–432.
20. Hutter, M.; Schwabe, P. Multiprecision multiplication on AVR revisited. J. Cryptogr. Eng. 2015, 5, 201–214. [CrossRef]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).