applied sciences

Article

Memory Efficient Implementation of Modular Multiplication for 32-bit ARM Cortex-M4

Hwajeong Seo

Department of IT convergence Engineering, Hansung University, 116 Samseong-Yoro-16-Gil Seongbuk-gu, Seoul 136-792, Korea; hwajeong@hansung.ac.kr

Received:12 January 2020; Accepted: 20 February 2020; Published: 24 February 2020

Abstract: In this paper, we present scalable multi-precision multiplication implementation and scalable multi-precision squaring implementation for 32-bit ARM Cortex-M4 microcontrollers. For efficient computation and scalable functionality, we present optimized Multiplication and ACcumulation (MAC) techniques for the target microcontrollers. In particular, we present the 64-bit wise MAC operation with the Unsigned Long Multiply with Accumulate Accumulate (`UMAAL`) instruction. The MAC is used to perform column-wise multiplication/squaring (i.e., product-scanning) with general-purpose registers in an optimal way. Second, the squaring algorithm is further optimized through an efficient doubling routine together with an optimized product-scanning method. Finally, the proposed implementations achieved a very small memory footprint and high scalability to cover algorityms ranging from well-known public key cryptography (i.e., Rivest–Shamir–Adleman (RSA) and Elliptic Curve Cryptography (ECC)) to post-quantum cryptography (i.e., Supersingular Isogeny Key Encapsulation (SIKE)). All SIKE round 2 protocols were evaluated with the proposed modular reduction implementations. The results demonstrate that the scalable implementation can achieve the smallest code size together with a reasonable performance.

Keywords: multi-precision multiplication; multi-precision squaring; public key cryptography; ARM Cortex-M4; memory-efficient implementation

1. Introduction

The implementation of cryptographic algorithms on an embedded device is more challenging than on personal computers due to the limited resources (e.g., low frequency, basic instruction set, and small RAM and ROM). Thus, cryptographic implementors have to carefully redesign or make specific optimizations of existing algorithms to fit such scenarios. In general, a lightweight implementation of a cryptographic algorithm on embedded devices should satisfy the following implementation requirements [1–3]: achieving high performance, having small code size, and supplying scalability to an arbitrary length. In recent decades, a number of works have improved the implementations of public key cryptography on microcontrollers. One of the milestone works was carried out by Gura et al. [4], who proposed the hybrid-scanning based multiplication. Compact implementations of Rivest–Shamir–Adleman (RSA) and Elliptic Curve Cryptography (ECC) on 8-bit Alf and Vegard's RISC processor (AVR) embedded processors, such as TinyECC [5], Relic [6], Networking and Cryptography library (NaCl) [7], and Montgomery and Twisted Edward (MoTE)-ECC [8] were also investigated.

In very recent years, the Advanced RISC Machine (ARM) company released a low-cost 32-bit ARM processor, called ARM Cortex-M4, in response to customer requests. The key characteristics of ARM's Cortex-M4 are a lower cost with a higher productivity than others. The embedded processor has a significantly small chip area, low energy consumption, and an optimal code footprint. These advanced capabilities are able to achieve a high performance at a low price point for various IoT applications, such as smart metering, motor control, and domestic household appliances. Compared to implementations on typical 8-bit processors, the Cortex-M4 achieved much higher performance thanks to its advanced hardware architecture.

Groot provided a constant-time implementation of X25519 for the ARM Cortex-M4 architecture [9]. In particular, a reduced-radix representation (25-bit or 26-bit) and refined Karatsuba multiplication are used for optimized implementation. Santis and Sigl implemented the Curve25519 on ARM Cortex-M4 microcontrollers [10]. The Karatsuba algorithm in a two-level subtractive is utilized for large integer multiplication. This approach replaces 256-bit wise multiplication into nine 64-bit wise multiplication. Fujii and Aranha implemented the integer multiplication with an operand-caching method and Unsigned Long Multiply with Accumulate Accumulate (`UMAAL`) instruction [11]. This work fully utilized the `UMAAL` instructions to achieve a high performance.

In CHES'19, the fastest implementation of Curve25519 is suggested by Haase and Labrique [12]. The hand-optimized operand-scanning method is efficiently implemented on the ARM Cortex-M4 microcontroller. For that reason, the utilization of the register is highly optimized. The first Supersingular Isogeny Key Encapsulation (SIKE) implementation is suggested by Koppermann et al. [13]. The product-scanning and Karatsuba methods are utilized to improve the Supersingular Isogeny Diffie-Hellman key exchange (SIDH). However, the implementation of SIDHp751 requires 18 s to exchange the keys. In [14], they implemented integer multiplication with operand caching (in `UMAAL`) and pipeline-friendly programming. The results show that SIKEp434 requires only 1.5 s.

While previous software implementations have focused on improving performance, they have paid relatively less attention to code size. In practice, most of the flash memory is used for application programs, and only a small footprint of the flash memory is used for cryptographic implementation. Many low-end ARM Cortex-M4 boards (e.g., Teensy 3.0, XMC4100, MAX32660, M481, etc.) have only 128–256 KB Flash memory. Furthermore, Internet of Things (IoT) devices must communicate with other devices with different protocols. For the Public Key Cryptography (PKC) implementation, there is pre-quantum cryptography and post-quantum cryptography.

During the transition and migration from pre-quantum to post-quantum, IoT devices should support both PKC algorithms according to the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) committee (i.e., hybrid PKC protocol) [15]. There are also several security levels (e.g., 128-bit brute-force attack, 192-bit, and 256-bit);, thus a number of PKC implementations should be considered. In particular, a lightweight PKC implementation can be achieved by optimized multi-precision multiplication implementation and optimized multi-precision squaring implementation. These methods are important for the deployment of cryptography for practical applications.

In Table 1, both multi-precision multiplication implementation and multi-precision squaring implementation are compared. The fastest performance is achieved by [14]. For the 256-bit ECC implementation (e.g., NIST P-256 and Curve25519), 772 bytes are required for the multiplication and squaring implementations. For the finite field operation, modular reduction is also needed, and it is usually a similar size of multiplication (i.e., 452 bytes) by using Montgomery reduction. To support SIKEp751, an additional 5,768 bytes are used for multiplication and squaring. Compared with other approaches, the proposed method is highly memory-efficient, requiring only 584 bytes. Moreover, the implementation is already parameterized, which supports all ECC, RSA, and SIKE protocols. By supporting all PKC protocols, the hybrid PKC protocol can be utilized. For this reason, the memory-efficient implementation can be a practical solution and the proposed method only satisfies this requirement.

Table 1. The comparison results of multi-precision multiplication implementation and multi-precision squaring implementation on 32-bit ARM Cortex-M4 microcontrollers in terms of the code size (in bytes), scalability, and highest speed. The Elliptic Curve Cryptography (ECC), Rivest–Shamir–Adleman (RSA), and Supersingular Isogeny Key Encapsulation (SIKE) symbols represent the targeted implementation with the modular arithmetic implementation (Letters U, L, and P indicate whether the works are implemented in unrolled, looped, or parameterized).

Methods	**Code Size**	**Scalability**	**SIKE**	**ECC**	**RSA**	**Speed Record**	**Hybrid Protocol**
			Multi-Precision Multiplication				
Groot † [9]	1284			✓			
Santis and Sigl † [10]	1264			✓			
Fujii and Aranha † [11]	622			✓			
Haase and Labrique [12]	452			✓			
Koppermann et al. [13]	–		✓				
Seo et al. [14]	452			✓		✓ (U)	
Seo et al. (SIKE) [14]	3,396		✓			✓ (U)	
This work	260	✓	✓	✓	✓	✓ (L,P)	✓
			Multi-precision squaring				
Groot † [9]	1168			✓			
Santis and Sigl † [10]	882					✓	
Fujii and Aranha † [11]	562			✓			
Haase and Labrique [12]	324			✓			
Seo et al. [14]	320			✓		✓ (U)	
Seo et al. (SIKE) [14]	2372		✓			✓ (U)	
This work	324	✓	✓	✓	✓	✓ (L,P)	✓

† symbol indicates implementations with fast reduction.

1.1. Comparison of CANS'19

Previous work in CANS'19 [14] successfully evaluated the SIKE round 2 schemes on 32-bit ARM Cortex-M4 microcontrollers. In order to accelerate the execution timing, modular multiplication, and squaring operations are optimized for specific parameters of SIKEp434, SIKEp503, and SIKEp751. Unlike previous works, we proposed scalable modular multiplication and squaring. The proposed implementation reduces the code size significantly and supports all parameters, including RSA, ECC, and SIKE, in single code. For the implementation of RSA, the proposed implementation only supports the RSA parameters, as described in Table 1. For instance, we implemented the SIKE round 2 schemes on 32-bit ARM Cortex-M4 microcontrollers in Section 3. The proposed implementation obtained the smallest code size and reasonably fast execution timing. Previous work in CANS'19 [14] requires 44,688 bytes while the proposed implementation requires 28,816 bytes for SIKEp751, which is a code reduction of 35.5%. Furthermore, we first implemented the SIKEp610 protocol on 32-bit ARM Cortex-M4 microcontrollers. This result first covers all SIKE protocols.

1.2. Research Contributions

In contrast to most of the previous implementations, which have focused primarily on the execution time, this study focused on optimizing the memory consumption of multi-precision multiplication implementation and multi-precision squaring implementation, without reducing the high performance. In particular, we present a MAC operation. The operation is used for the inner loop of target multiplication and squaring operations.

The proposed MAC routine fully employs general purpose registers and a $32 \times 32 + 32 + 32 \rightarrow$ 64-bit multiplier, also known as Unsigned Long Multiply with Accumulate Accumulate (`UMAAL`). Second, the squaring algorithm is further optimized through a dedicated doubling routine. Finally, the proposed implementations achieved a very small memory footprint and good scalability; thus, they can be used for RSA, ECC, and SIKE. We implemented all of the second round of SIKE protocols on 32-bit ARM Cortex-M4 microcontrollers. A reasonable execution timing can be achieved with the smallest code size.

The paper is written as follows. In Section 2, previous implementations of multiplication/squaring, target ARM Cortex-M4 microcontrollers, and SIKE Round 2 candidates are introduced. In Sections 2.4 and 2.5, implementations of the multi-precision multiplication method and multi-precision squaring method are presented. In Section 3, a case study of SIKE round 2 protocols is given. The proposed implementations are evaluated in Section 4. Conclusion is given in Section 5.

2. Background

2.1. Multi-Precision Multiplication and Squaring

A straightforward method to execute the large integer multiplication is row-wise multiplication (i.e., operand-scanning), which consists of a nested loop. One digit of an operand A (i.e., $a[i]$) of the inner loop is multiplied with all digits of the second operand B, while the pointer of operand A points to the next digit of A, in the outer loop. An alternative method is column-wise multiplication [16]. The product-scanning method consists of two inner loops. The first loop performs the lower part of the result. Afterward, the second loop performs the upper part of the result. An important operation in each inner loop is called the Multiply–ACcumulate (MAC) operation, which performs a form of $C \leftarrow C + a[i] \times b[j]$, where C is three digits of the intermediate results. The intermediate results are not stored or loaded to/from the memory. Carry propagation is simply achieved by a register-copy operation. In recent years, many works have improved the product-scanning method by re-ordering the inner loop and using new instructions [4,17,18].

A squaring operation is a unique case of multi-precision multiplication, where the multiplier and multiplicand are the identical operand (e.g., $A \times A$). For multi-precision squaring, partial products of the form ($a[i] \times a[j]$ with $i \neq j$) are doubled based on the fact that $a[i] \times a[j] = a[j] \times a[i]$, while $a[i] \times a[j]$ appears just once for $i = j$. Thus, squaring requires less `mul` instructions or MAC operations than multiplication. Some previous implementations for squaring have been optimized based on this symmetric feature on embedded processors. The lazy-doubling method optimizes the doubling operation by performing the doubling column-wise [19]. Similarly, many works have focused on developing an optimized doubling method for target microcontrollers [20,21].

2.2. ARM Cortex-M4 Microcontroller

32-bit ARM Cortex-M4 embedded microprocessors are famous low-end microcontrollers. The processor is selected for benchmark target processor of NIST post-quantum cryptography. Recently, one benchmark framework (`pqm4`) has also been implemented on Cortex-M4 [22]. The embedded microprocessor supports an optimal multiplication instruction (i.e., `UMAAL`). The instruction is multiplication-accumulation with two 32-bit values.

2.3. SIKE Round 2

We introduce the SIKE standard and key exchange protocol. For better understanding, we recommend to refer to [23,24]. The SIKE standard is using a transformation by [25]. This ensures the supersingular isogeny Public Key Encryption (PKE) protocol [23]. This is Key Encapsulation Mechanism (KEM), which is secure against the static key vulnerability of the key exchange protocol [26]. The SIKE protocol is a NIST PQC round 2 candidate [27]. SIKE has relatively small public keys and ciphertext.

2.3.1. Public Parameters

The SIKE is over a prime ($p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$). For better performance, $\ell_A = 2, \ell_B = 3$, and $f = 1$ are fixed. The prime of SIKE is $p = 2^{e_A} \cdot 3^{e_B} - 1$. The beginning supersingular elliptic curve $E_0/\mathbb{F}_{p^2}$: $y^2 = x^3 + x$ with cardinality equal to $(2^{e_A} \cdot 3^{e_B})^2$, along with base points $\langle P_A, Q_A \rangle = E_0[2^{e_A}]$ and $\langle P_B, Q_B \rangle = E_0[3^{e_B}]$ are defined as public parameters.

2.3.2. Key Encapsulation Mechanism

KEM consists of three parts: Alice's key generation, Bob's key encapsulation, and Alice's key decapsulation. Figure 1 describes the KEM in detail.

Alice		**Bob**
Key generation:		
$\texttt{pk}_A = [E_A, \phi_A(P_B), \phi_A(Q_B)]$		
$s \in_R \{0,1\}^t$		
		Encapsulation:
		$m \in_R \{0,1\}^t$
		$r = H_1(m \parallel \texttt{pk}_A)$
		$\texttt{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$
		$j = j(E_{BA})$
		$c = (c_0, c_1) = (\texttt{pk}_B(r), H_2(j) \oplus m)$
		$K = H_3(m \parallel c)$
	$\xleftarrow{(c_0, c_1)}$	
Decapsulation:		
$j = j(E_{AB})$		
$m' = c_1 \oplus H_2(j)$		
$r' = H_1(m' \parallel \texttt{pk}_A)$		
If $(\texttt{pk}_B(r') = c_0) \rightarrow K = H_3(m' \parallel c)$		
If $(\texttt{pk}_B(r') \neq c_0) \rightarrow K = H_3(s \parallel c)$		

Figure 1. SIKE mechanism.

Key Generation

Alice chooses an random integer $\text{sk}_A \in \mathbb{Z}/2^{e_A}\mathbb{Z}$ and by applying an isogeny $\phi_A : E_0 \rightarrow E_A$ with kernel $R_A := \langle P_A + [\text{sk}_A]Q_A \rangle$ to the base points $\{P_B, Q_B\}$, computes her public key $\text{pk}_A = [E_A, \phi_A(P_B), \phi_A(Q_B)]$. Moreover, Alice generates a t-bit (the implementation parameter defines the t value) random sequence $s \in_R \{0,1\}^t$.

Encapsulation

Bob generates a t-bit random message m $\in_R \{0,1\}^t$. Afterward, Bob concatenates the message with her public key pk_A. Bob generates an e_B-bit hash result r with `cSHAKE256` hash function H_1 and input $(m \parallel \text{pk}_A)$. With r, Bob executes a secret isogeny $\phi_B : E0 \rightarrow EB$ to base points $\{P_A, Q_A\}$. Then, Bob forms his public key $\text{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$. Bob also performs the common j-invariant of curve E_{BA} by using another isogeny $\phi'_B : EA \rightarrow E_{BA}$ with Alice's public key. Finally Bob generates a ciphertext $c = (c_0, c_1)$, such that:

$$c = (c_0, c_1) = (\text{pk}_B(r), H_2(j(E_{BA})) \oplus m),$$

where H_2 is generated with a `cSHAKE256` hash function. This generates an arbitrary-length of output with a defined initialization parameter. Lastly, Bob generates the shared secret $K = H_3(m \parallel c)$. Afterward, Bob sends the ciphertext (c) to Alice.

Decapsulation

With the ciphertext (c), Alice performs the common j-invariant of E_{AB} by using her secret isogeny to E_B. Alice executes $m' = c_1 \oplus H_2(j(E_{AB}))$ and $r' = H_1(m \parallel \text{pk}_A)$. Lastly, Alice validates Bob's public key by performing $\text{pk}_B(r')$ and comparing the value with c_0. Alice generates the secret shared value $K = H_3(m' \parallel c)$ when the public key is correct. Otherwise, Alice generates a random value $K = H_3(c \parallel s)$ to be secure against active attacks.

In 2019, the candidates for the second round of post-quantum cryptography competition is selected. The round 2 of SIKE protocol [28] is different from the first version. These changes are outlined as follows:

- Two new parameter sets for NIST security levels 1 and 3 have been added (i.e., SIKEp434 and SIKEp610).
- One parameter set (i.e., SIKEp964) has been removed.
- The security categories for parameter sets have been adjusted upward (i.e., the NIST security levels of SIKEp503 and SIKEp751 are changed to 2 and 5, respectively.).
- The starting curve has been changed.
- A public key compression has been implemented.

2.4. Multi-Precision Multiplication

For both outer and inner loops, the product-scanning method is utilized. In Figure 2, detailed descriptions are given. The left part is for the outer loop, and the right part is for the inner loop. One block in the outer loop consists of two word-multiplication, indicated by the dashed boxes and colors in the right part of Figure 2. At the beginning, two words of operand A (labeled a_0 and a_1 in Figure 2) along with two words of B (namely b_0 and b_1) are loaded from the RAM.

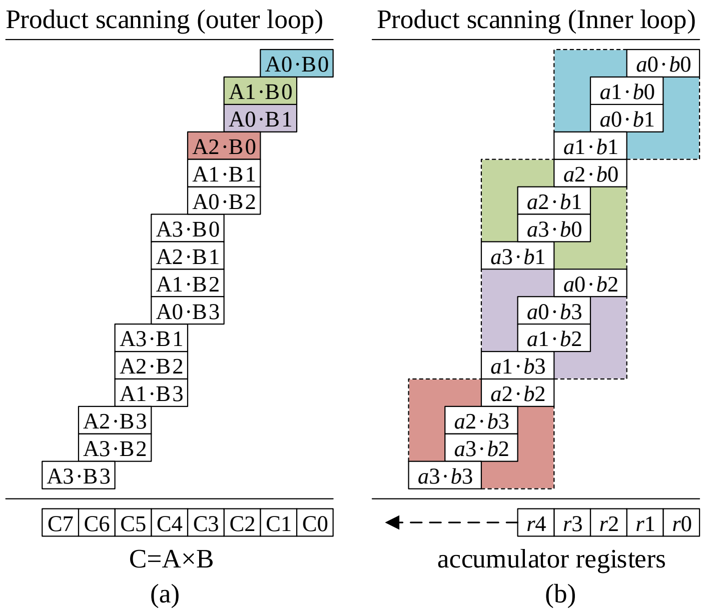

Figure 2. The proposed product scanning multiplication. (**a**) outer loop, an example of 128-bit multiplication. (**b**) inner loop, ($A0 \cdot B0$, $A1 \cdot B0$, $A0 \cdot B1$, $A2 \cdot B0$).

The 64-bit product of a_0 and b_0 is performed and accumulated to two registers by using the `UMAAL` instruction. Afterward, the product of $a_1 \cdot b_0$ is performed and the product of the two words is added into the accumulator registers. The carry values from this operation can be performed without other side effects. Thereafter, we multiply a_0 by b_1, and add the resulting 64-bit product of $a_0 \cdot b_1$ to registers. The result of the higher word is stored in a temporal register before accumulation to avoid overflows.

After the last product of the first block (i.e., $a_0 \cdot b_0$), we add the products with values in temporary registers to the three accumulator registers. The carry bit is finally propagated. The execution of the first block in Figure 2 executes four `UMAAL`, three `SUB`, and three `ADD` instructions, respectively. These instructions require 10 clock cycles. In Algorithm 1, the source code of the proposed implementation is given. In lines 1 to 3, the operands are loaded to the registers and the address pointer is corrected. In lines 4 to 9, the multiplication and accumulation routine is performed. In particular, registers `R1` and R2 are cleared with SUB instructions. In lines 10 to 12, the intermediate results are accumulated to the registers. The subsequent blocks are executed by following the computation of the first block.

Algorithm 1 Multiply–ACcumulate (MAC) operation for multiplication.

```
Require: operand pointers R0, R12
Ensure: results {R8, R4, R5, R6, R7}
 1: LDM R12!, {R10-R11}
 2: LDM R0!, {R2-R3}
 3: SUB R0, #16

 4: SUB R1, R1, R1
 5: UMAAL R8, R1, R10, R2          {Low-Low}
 6: UMAAL R4, R1, R11, R2          {High-Low}

 7: SUB R2, R2, R2
 8: UMAAL R4, R2, R10, R3          {Low-High}
 9: UMAAL R5, R1, R11, R3          {High-High}

10: ADD R5, R5, R2                 {Low-High}
11: ADC R6, R6, R1                 {High-High}
12: ADC R7, R7, R3
```

In Algorithm 2, full rounds of integer multiplication are given. In line 1, the accumulation buffer is initialized. The multiplication is divided into two steps, which are lower result (i.e., lines 2–9) and higher result (i.e., lines 10–18), respectively. The core MAC operation is implemented by following Algorithm 1.

Algorithm 2 64-bit wise scalable product-scanning for multiplication.

```
Require: operands ($a$ and $b$) and $len$ (operand length/word where the word is 64-bit)
Ensure: results ($r$)
 1: $accumulation = 0$
 2: for $i = 0$ to $len - 1$ by $1$ do
 3:     for $j = 0$ to $i$ by $1$ do
 4:         $product = a[j] \times b[i-j]$                        {Algorithm 1}
 5:         $accumulation = accumulation + product$
 6:     end for
 7:     $r[i] = accumulation \bmod 2^{64}$
 8:     $accumulation = accumulation \gg 64$
 9: end for
10: for $i = len$ to $2 \times len - 2$ by $1$ do
11:     for $j = i - len + 1$ to $len - 1$ by $1$ do
12:         $product = a[j] \times b[i-j]$                        {Algorithm 1}
13:         $accumulation = accumulation + product$
14:     end for
15:     $r[i] = accumulation \bmod 2^{64}$
16:     $accumulation = accumulation \gg 64$
17: end for
18: $r[i] = accumulation \bmod 2^{64}$
```

2.5. Multi-Precision Squaring

A novel method for implementing the multi-precision squaring method, called the "doubling and MAC" method, is proposed. The squaring is implemented by following the structure of the column-wise method for the "outer algorithm". For the "inner algorithm", a combination of the proposed product-scanning and doubling and MAC method is performed.

The proposed technique optimizes the number of ADD (resp. ADC) instructions through rearranging the sequence of the UMAAL operation. An example of 256-bit squaring is shown in Figure 3. The left part describes the outer loop of squaring. It consists of three parts (inside the dashed box), which are exactly in line with the three loops of squaring. The middle part and the right part show the inner loop of squaring. The middle part is used to calculate $a_i \cdot a_j$ for $i \neq j$, similar to the procedure that we presented in Section 2.4 for multiplication. The proposed doubling and MAC for the computation of $A_i \cdot A_i$ is given in following section.

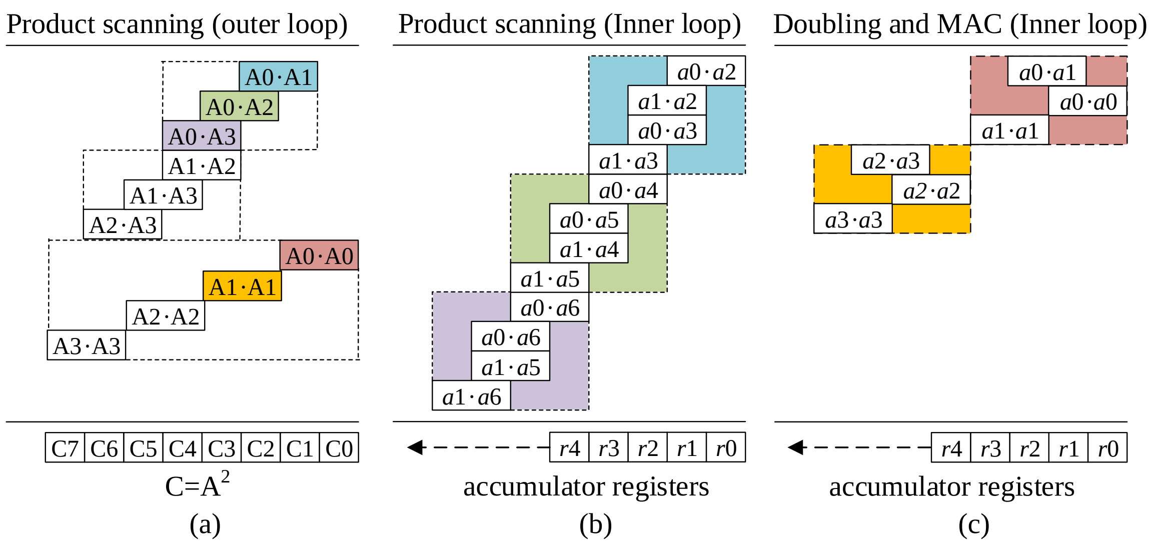

Figure 3. The proposed product-scanning squaring. (**a**) outer loop, an example of 128-bit squaring. (**b**) inner loop, product scanning, ($A0 \cdot A0$, $A0 \cdot A2$, $A0 \cdot A3$). (**c**) inner loop, doubling and MAC, ($A0 \cdot A0$, $A1 \cdot A1$).

In Figure 3, the procedure of doubling and MAC can be split into two blocks as represented by dashed boxes. Taking the computation of $A_0 \cdot A_0$ (marked in red) as an example, the 64-bit operand A_0 can be represented as (a_0 and a_1), where a_i is one word long.

First, we load the intermediate results computed by the middle part of Figure 3 from the memory to the registers. Afterward, A_0 is loaded into two registers which are labeled a_0 and a_1. We first perform the multiplication of $a_0 \cdot a_1$ and accumulate the 64-bit product to three intermediate result registers. The accumulated intermediate results are doubled at once. Next, we perform two-word multiplication ($a_0 \cdot a_0$ and $a_1 \cdot a_1$) and the results are accumulated to the doubled results. During accumulation, we catch the carry value and store it in the temporal register.

The carry value is used in the next doubling and MAC routine. In total, the proposed doubling and MAC method costs 21 clock cycles, including 3 `UMAAL`, 12 `ADD` (resp. `ADC`), 4 `MOV` and 1 `SUB` instructions. The source code can be found in Algorithm 3. In line 2, the operands are loaded to registers (`R8, R9`). In line 3, intermediate results are loaded from memory to registers (`R2, R3, R4, and R5`). In lines 4 and 5, two registers are cleared. In lines 6 to 9, the partial product ($a_i \cdot a_j$ for $i = j$) is performed and accumulated to the intermediate results. In lines 10 to 14, the intermediate results are doubled. In lines 15 to 17, the two-word multiplication ($a_i \cdot a_j$ for $i \neq j$) is performed. In lines 18 to 23, the results are accumulated to the doubled results. During accumulation, the carry value is stored in the `R7` register. In lines 24 to 25, the intermediate results are stored in the memory. Finally, in lines 26 to 27, the loop count is measured, and the program counter is calculated depending on the loop condition.

In Algorithm 4, full rounds of squaring implementation are given. In lines 1–21, similar to the integer multiplication, partial products ($a[i] \times a[j]$ where $i \neq j$) are performed by using Algorithm 1. In lines 22–34, the remaining part ($a[i] \times a[j]$ where $i = j$) is performed with doubling and MAC routine given in Algorithm 3.

Algorithm 3 Doubling and MAC implementation with a one-way carry-catcher technique.

Require: result pointer `R11`, operand pointer `R12`, loop counter `R14`, carry-catcher `R1, R7`
Ensure: results `{R2, R3, R4, R5, R6}`

```
 1: LOOP:
 2: LDM R12!, {R8-R9}
 3: LDM R11!, {R2-R5}
 4: MOV R1, #0
 5: MOV R6, #0

 6: UMAAL R3, R0, R8, R9          {Low-High}
 7: ADD R4, R0                    {Low-High}
 8: ADC R5, R1
 9: ADC R6, R1

10: ADD R2, R2                    {Doubling}
11: ADC R3, R3
12: ADC R4, R4
13: ADC R5, R5
14: ADC R6, R6

15: MOV R0, #0
16: UMAAL R2, R1, R8, R8          {Low-Low}
17: UMAAL R4, R0, R9, R9          {High-High}

18: ADD R2, R7

    {Carry-Catcher #1}
19: MOV R7, #0
20: ADC R3, R1                    {Low-Low}
21: ADC R4, R7
22: ADC R5, R0                    {High-High}
23: ADC R7, R6

24: SUB R11, #16
25: STM R11!, {R2-R5}

26: CMP R14, R11
27: BHS LOOP
```

Algorithm 4 64-bit wise scalable product-scanning for squaring.

Require: operand (a) and len (operand length/word where word is 64-bit)
Ensure: results (r)

```
 1: $accumulation = 0$
 2: for $i = 0$ to $len - 1$ by $1$ do
 3:   for $j = 0$ to $i$ by $1$ do
 4:     if $i \neq j$ then
 5:       $product = a[j] \times a[i - j]$                    {Algorithm 1}
 6:       $accumulation = accumulation + product$
 7:     end if
 8:   end for
 9:   $r[i] = accumulation \bmod 2^{64}$
10:   $accumulation = accumulation \gg 64$
11: end for
12: for $i = len$ to $2 \times len - 2$ by $1$ do
13:   for $j = i - len + 1$ to $len - 1$ by $1$ do
14:     if $i \neq j$ then
15:       $product = a[j] \times a[i - j]$                    {Algorithm 1}
16:       $accumulation = accumulation + product$
17:     end if
18:   end for
19:   $r[i] = accumulation \bmod 2^{64}$
20:   $accumulation = accumulation \gg 64$
21: end for
22: $accumulation = 0$
23: for $i = 0$ to $2 \times len - 1$ by $1$ do
24:   if $i/2 = 0$ then
25:     $product = a[i] \times a[i]$                          {Algorithm 3}
26:   else
27:     $product = 0$
28:   end if
29:   $double = 2 \times r[i]$
30:   $accumulation = accumulation + double + product$
31:   $r[i] = accumulation \bmod 2^{64}$
32:   $accumulation = accumulation \gg 64$
33: end for
34: $r[i] = accumulation \bmod 2^{64}$
```

3. Scalable Montgomery Reduction for All SIKE Round 2 on Cortex-M4

The parameter sets (SIKEp434, SIKEp503, SIKEp610, and SIKEp751) are selected as the SIKE round-2 protocol [28,29]. For high performance, the Montgomery reduction method is used [30]. The Montgomery reduction replaces the expensive inversion operation into a relatively cheap operation (i.e., multiplication). The detailed descriptions for the Montgomery reduction are given in Algorithm 5.

We evaluated the all-SIKE protocols with the proposed multi-precision multiplication operation and multi-precision squaring operation. For the Montgomery reduction, we used a scalable operand-scanning method. To support all-SIKE protocols, the internal word size is set to 32-bit wise. We divided the n-word operand scanning into three steps, including initialization, a middle round, and finalization. The initialization calculates one partial product first. The intermediate result is stored in the memory, which is a quotient value. The middle round performs the $(n-1)$-word partial products. The result is accumulated to the intermediate result. Finally, the last word is multiplied and added to the intermediate results. Detailed descriptions of the SIKEp434 case are given in Figure 4.

Algorithm 5 Montgomery reduction.

```
Require: Modulus $m$, Montgomery radix $r > m$, operand $c \in [0, m^2 - 1]$, and constant $m' = -m^{-1} \bmod r$
Ensure: Montgomery product $z = \mathrm{MonRed}(c, r) = c \cdot r^{-1} \bmod m$
1: $q \leftarrow c \cdot m' \bmod r$
2: $z \leftarrow (c + q \cdot m)/r$
3: if $z \geq m$ then $z \leftarrow z - m$
4: return $z$
```

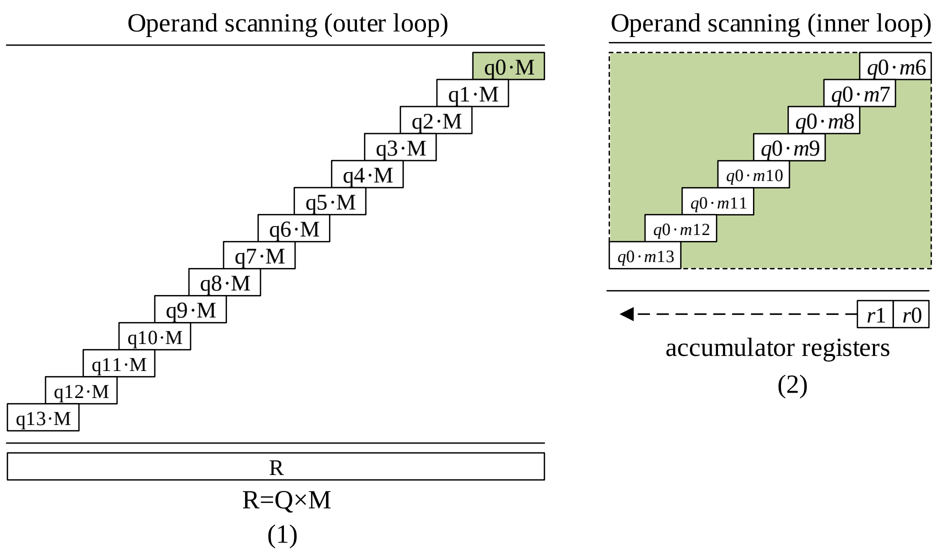

Figure 4. Proposed operand scanning multiplication for Montgomery reduction. **Left part**: outer loop, an example of SIKEp434. **Right part**: inner loop, $(q0 \cdot M)$.

The modulus (M) of SIKEp434 is multiplied by the quotient in a 32-bit wise (Q; q0–q13). Since the lower part of the modulus (i.e., m0–m5) is zero, only remaining parts (m6–m13) are multiplied. During the inner loop, only two registers are used to maintain the intermediate results. The inner loop of operand scanning is given in Algorithm 6. In line 1, one operand Q is loaded to the register (`R8`). In line 2, the intermediate result is loaded to the register (`R9`). In line 3, the 32-bit wise MAC is performed. In line 4, the result is stored in the memory.

Algorithm 6 Inner loop of operand scanning.

Require: operand pointers R1, R4
Ensure: results {R9, R10}

```
LDM R1!, {R8}
LDR R9, [R4, #0]
UMAAL R9, R10, R7, R8
STM R4!, {R9}
```

In Algorithm 7, 32-bit wise scalable operand-scanning for Montgomery reduction is described. All operations are performed in 32-bit wise. In particular, the inner loop (i.e., line 4) is performed with the Algorithm 6.

Algorithm 7 32-bit wise scalable operand-scanning for Montgomery reduction.

Require: modulus m, constant m', and len (operand length/word where the word is 32-bit)
Ensure: results (r)

```
1:  for $i = 0$ to $len - 1$ by 1 do
2:      $prod = 0$
3:      $q = r[0] \times m' \bmod 2^{32}$
4:      $prod = m[0] \times q$                    {Algorithm 5}
5:      $prod += r[0]$
6:      $prod = prod \gg 32$
7:      for $j = 1$ to $len - 1$ by 1 do
8:          $prod += m[j] \times q$
9:          $prod += r[0]$
10:         $r[j-1] = prod$
11:         $prod = prod \gg 32$
12:     end for
13:     $prod += r[j]$
14:     $r[j-1] = prod$
15:     $r[j] = r[j+1] + prod \gg 32$
16: end for
```

4. Performance Evaluation and Comparison

4.1. Evaluation of Modular Arithmetic

The 32-bit ARM Cortex-M4 processor is evaluated with the STM32F4 Discovery board. The board supports 1 MB of Flash and 192 KB of RAM. Table 2 compares the proposed implementation with other works in terms of the execution timing. The work by Seo et al. focused on high speed optimizations with the operand caching method and the sliding block doubling method. The proposed 256-bit multiplication and squaring operations are slower than those proposed by Seo et al. by 67% and 72%, respectively, and the 768-bit multiplication and squaring operations are slower than Seo et al. by 57% and 56%, respectively.

The performance is reasonable as the 32-bit ARM Cortex-M4 microcontrollers support 168 MHz, which means that Curve25519 and SIKEp434 can be still performed within seconds. In terms of code size, our implementation only requires 584 bytes (260 bytes for multiplication and 324 bytes for squaring, respectively). This requires 75% and 10% of the code size used by Seo et al., which directly saves 0.2–5 KB of code size for PKC implementations.

In terms of long integer multiplication, previous works did not explore the implementations. This means that previous implementations cannot cover RSA implementations. For this reason, we evaluated the proposed implementations and compared the results to the estimated results by Seo et al. For the RSA3072 encryption, the 3072-bit multiplication and squaring implementations based on Seo et al. require about 54 KB and 38 KB, respectively. Furthermore, the fast RSA3072 decryption based on the Chinese-Remainder Theorem (CRT) needs 1536-bit multiplication and squaring implementations (13 KB for multiplication and 9 KB for squaring). For this reason, the unrolled RSA implementation cannot be used due to the code size. On the other hand, the proposed method still maintains a small code size (260 bytes for multiplication and 324 bytes for squaring). The results show that our method only provides all PKC options (i.e., ECC, RSA, and SIKE) on ARM Cortex-M4 microcontrollers.

It is also interesting to compare parameterized implementations on other low-cost platforms (e.g., 8-bit AVR processors). Compared with previous parameterized implementations over the AVR processor [1], our implementation has several distinguishing features. In terms of the MAC algorithm, we introduced an optimized MAC method, which is specialized for the new `UMAAL` multiplier of target processor. The one-way carry-catcher method for integrated doubling and MAC operations introduces further optimizations of memory access.

Table 2. The comparison results of the code size (bytes) and execution time (clock cycles) of different multi-precision multiplication implementations and multi-precision squaring implementations for operands ranging from 256 to 3072 bits.

Approach	Code Size	256	512	768	1536	3072
Public Key Cryptography		**ECC**	**SIKE**	**SIKE**	**RSA**	**RSA**
Multi-Precision Multiplication						
Groot † [9]	1284	631	–	–	–	–
Santis and Sigl † [10]	1264	546	–	–	–	–
Fujii and Aranha † [11]	622	239	–	–	–	–
Haase and Labrique [12]	452	212	–	–	–	–
Koppermann et al. [13]	–	–	–	4,319	–	–
Seo et al. [14]	452	196	–	–	–	–
Seo et al. (SIKE) [14]	3,396	–	–	1556	–	–
This work	260	608	1816	3696	13,369	50,857
Multi-precision squaring						
Groot † [9]	1168	563	–	–	–	–
Santis and Sigl † [10]	882	362	–	–	–	–
Fujii and Aranha † [11]	562	218	–	–	–	–
Haase and Labrique [12]	324	141	–	–	–	–
Seo et al. [14]	320	136	–	–	–	–
Seo et al. (SIKE) [14]	2372	–	–	1054	–	–
This work	324	493	1285	2397	7654	26,806

† Including fast reduction.

4.2. Evaluation of SIKE

Table 3 compares the execution time and ROM of the SIKE round 2 protocols with other works. A 32-bit ARM Cortex-M4 processor is evaluated with the STM32F4 Discovery board. The board supports 1 MB of Flash and 192 KB of RAM. For the fair benchmark test of SIKE, the well-known `pqm4` library is uploaded to the board and the actual system clock cycles are obtained. This is a well-known benchmark framework for post-quantum cryptography.

Previous works by Seo et al. mainly focused on the speed optimization [14], while the proposed implementation targets the minimum code size. The previous SIKEp434 implementation achieved 252×10^6 clock cycles, while the proposed SIKEp434 implementation achieved 469×10^6 clock cycles. The execution timing is translated into throughput. As we evaluated the performance at 168 MHz, the full key exchange of SIKEp434 is performed within 2.79 s on low-end microcontrollers, which is reasonably fast. By considering the speed and size trade-off, the performance of the SIKEp434 protocol is 46% (252×10^6 cc vs. 469×10^6 cc) slower than that achieved in the previous work, but we reduced the code size by 13% in this case (33,528 bytes vs. 29,176 bytes). The enhancement of execution timing is similar for SIKEp751, but the difference in code size between the speed version and the size version in greater. The previous SIKEp751 implementation requires 44,688 bytes, while the proposed SIKEp751 implementation requires 28,816 bytes.

Table 3. Comparison of SIKE round 2 schemes on ARM Cortex-M4 microcontrollers. Timings are measured in clock cycles. Code size is reported in terms of bytes. Throughput is reported in terms of seconds per operation at 168 MHz.

Implementation	**Language**	**Timings [*cc*]**		**Timings [$cc \times 10^6$]**				**Throughput**	**ROM**	**Optimization**
		$\mathbb{F}_p$ mul	**$\mathbb{F}_p$ sqr**	**KeyGen**	**Encaps**	**Decaps**	**Total**	**[secs/op]**	**[bytes]**	
SIKEp434										
Seo et al. [14]	ASM	1110	981	74	122	130	252	1.50	33,528	Speed
This work	ASM	3147	2693	139	227	242	469	2.79	29,176	Size
SIKEp503										
Seo et al. [14]	ASM	1333	1139	104	172	183	355	2.11	37,912	Speed
This work	ASM	3888	3280	197	326	346	672	4.00	30,488	Size
SIKEp610										
This work	ASM	5651	4639	340	631	634	1265	7.53	27,432	Size
SIKEp751										
Seo et al. [14]	ASM	2744	2242	282	455	491	946	5.63	44,688	Speed
This work	ASM	7693	6228	587	953	1022	1975	11.76	28,816	Size

We also first implemented the SIKEp610 protocols on ARM Cortex-M4 processors, and the execution timing and code size achieved reasonable results. The other strength of scalable implementation is definitely scalability. The SIKE implementations share the majority of operations except special parameters and finite field operations. All SIKE implementations, including SIKEp434, SIKEp503, SIKEp610, and SIKEp751 with scalable multiplication and squaring may require around 40 KB. However, all SIKE implementations with speed optimized multiplication and squaring may require around 100 KB.

The system requirements in terms of speed (i.e., throughput), resources (i.e., code size), and energy consumption are determined by the architectural choices for a given application. Table 4 compares the practicality of implementations for public key cryptography. Previous works only focused on the speed optimization for one targeted PKC protocol. However, real world applications require multiple cryptography implementations to communicate with heterogeneous systems with multiple PKC schemes and security levels. Furthermore, for modular arithmetic of RSA implementations, speed optimized implementation requires a huge code size (See Table 1).

Table 4. Comparison of practical implementations.

Public Key Cryptography	**ECC**	**All ECC**	**SIKE**	**All SIKE**	**RSA**	**All RSA**
Groot † [9]	√	–	√	–	–	–
Santis and Sigl † [10]	√	–	√	–	–	–
Fujii and Aranha † [11]	√	–	√	–	–	–
Haase and Labrique [12]	√	–	√	–	–	–
Koppermann et al. [13]	√	–	√	–	–	–
Seo et al. [14]	√	–	√	–	–	–
This work	√	√	√	√	√	√

† Including fast reduction.

On the other hand, the proposed modular arithmetic implementation supports all RSA, ECC, and SIKE with the smallest code size (e.g., 584 bytes) and reasonable execution timing. In terms of the energy consumption, the proposed method actually consumes more energy than speed optimized implementation due to the high latency (i.e., long execution timing or long working time), which consumes high energy [31–33]. However, the proposed method is a suitable solution when considering the reasonable code size in real-world implementations. The importance of energy consumption for PKC is relatively lower than block cipher as PKC only needs one time before the secure communication as a secure key exchange (i.e., Diffie-Hellman). For this reason, the energy consumption metric of PKC implementation is of relatively lower priority than the code size in a real-world setting.

5. Conclusions

In this paper, we presented scalable implementations of multi-precision multiplication, squaring, and the Montgomery algorithm on the 32-bit ARM Cortex-M4. The implementation emphasizes reducing code size and providing scalability with practically fast speed. We proposed several novel techniques to further boost the implementations on low-cost 32-bit ARM processors, including MAC as well as doubling and MAC techniques. Our implementation on the 32-bit ARM platform highlights the practical benefits of the proposed methods. The multiplication and squaring implementations on the ARM Cortex-M4 require execution times of 608 and 493 clock cycles for 256-bit, and 50,857 and 26,806 clock cycles for 3072-bit, respectively. Even though our implementation is slower than the unrolled implementation, it only requires 10% of the memory footprint for the SIKEp751 case. Furthermore, the proposed multiplication and squaring require only 584 bytes for code size, which is perfectly suitable for the PKC (e.g., RSA, ECC, and SIKE) implementations on low-cost processors. The proposed implementations are fully parameterized and are based on a constant-time solution. An operand of any length can be supported with a single implementation. We also evaluated the all-SIKE round 2 protocols on the target microcontrollers. The results show that a reasonable execution timing can be achieved with a very small code size.

Funding: This work was partly supported as part of Military Crypto Research Center(UD170109ED) funded by Defense Acquisition Program Administration(DAPA) and Agency for Defense Development(ADD) and this work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services).

Conflicts of Interest: The author declare no conflicts of interest.

References

1. Liu, Z.; Seo, H.; Großschädl, J.; Kim, H. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In *Information and Communications Security*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 158–175.
2. Seo, H. Compact software implementation of public-key cryptography on MSP430X. *ACM Trans. Embed. Comput. Syst.* **2018**, *17*, 66. [CrossRef]
3. Liu, Z.; Seo, H.; Castiglione, A.; Choo, K.K.R.; Kim, H. Memory-efficient implementation of elliptic curve cryptography for the Internet-of-Things. *IEEE Trans. Dependable Secur. Comput.* **2018**, *16*, 521–529. [CrossRef]
4. Gura, N.; Patel, A.; Wander, A.; Eberle, H.; Shantz, S.C. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems—CHES 2004*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 119–132.
5. Liu, A.; Ning, P. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN'08), St. Louis, MO, USA, 22–24 April 2008; IEEE: Hoboken, NJ, USA, 2008; pp. 245–256.
6. Aranha, D.F.; Gouvêa, C.P.L. RELIC is an Efficient LIbrary for Cryptography. Available online: https://github.com/relic-toolkit/relic (accessed on 18 February 2020).
7. Hutter, M.; Schwabe, P. NaCl on 8-bit AVR microcontrollers. In *Progress in Cryptology—AFRICACRYPT 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 156–172.
8. Liu, Z.; Wenger, E.; Großschädl, J. MoTE–ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In *Applied Cryptography and Network Security*. Springer: Berlin/Heidelberg, Germany, 2014, pp. 361–379.
9. De Groot, W. A Performance Study of X25519 on Cortex-M3 and M4. Master's Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2015.

10. De Santis, F.; Sigl, G. Towards side-channel protected X25519 on ARM Cortex-M4 processors. In Proceedings of the SPEED-B—Software Performance Enhancement for Encryption and Decryption, and Benchmarking, Utrecht, The Netherlands, 19–21 October 2016.
11. Fujii, H.; Aranha, D.F. Curve25519 for the Cortex-M4 and beyond. In *Progress in Cryptology–LATINCRYPT 2017*; Springer: Berlin/Heidelberg, Germany, 2017.
12. Haase, B.; Labrique, B. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2019**, *2019*, 1–48.
13. Koppermann, P.; Pop, E.; Heyszl, J.; Sigl, G. 18 Seconds to key exchange: Limitations of supersingular isogeny Diffie-Hellman on rmbedded devices. *IACR Cryptolog. ePrint Arch.* **2018**, *2018*, 932.
14. Seo, H.; Jalali, A.; Azarderakhsh, R. SIKE round 2 speed record on ARM Cortex-M4. In Proceedings of the The 18th International Conference on Cryptology And Network Security, Vienna, Austria, 25–27 October 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 39–60.
15. Moody, D. *Let's Get Ready to Rumble. The NIST PQC "Competition"*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2018.
16. Comba, P.G. Exponentiation cryptosystems on the IBM PC. *IBM Syst. J.* **1990**, *29*, 526–538. [CrossRef]
17. Hutter, M.; Wenger, E. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Cryptographic Hardware and Embedded Systems–CHES 2011*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 459–474.
18. Seo, H.; Kim, H. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In *Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 55–67.
19. Lee, Y.; Kim, I.H.; Park, Y. Improved multi-precision squaring for low-end RISC microcontrollers. *J. Syst. Softw.* **2013**, *86*, 60–71. [CrossRef]
20. Seo, H.; Liu, Z.; Choi, J.; Kim, H. Multi-precision squaring for public-key cryptography on embedded microprocessors. In Proceedings of the Progress in Cryptology—INDOCRYPT 2013, Mumbai, India, 7–10 December 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 227–243.
21. Seo, H.; Liu, Z.; Choi, J.; Kim, H. Optimized Karatsuba squaring on 8-bit AVR processors. *Secur. Commun. Netw.* **2015**, *8*, 3546–3554. [CrossRef]
22. Kannwischer, M.J.; Rijneveld, J.; Schwabe, P.; Stoffelen, K. PQM4: Post-quantum crypto library for the ARM Cortex-M4. Available online: https://github.com/mupq/pqm4 (accessed on 18 February 2020).
23. Jao, D.; Feo, L.D. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Proceedings of the Post-Quantum Cryptography, Taipei, Taiwan, 29 November–2 December 2011* ; Yang, B., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011, Volume 7071, pp. 19–34.
24. Azarderakhsh, R.; Campagna, M.; Costello, C.; Feo, L.D.; Hess, B.; Jalali, A.; Jao, D.; Koziel, B.; LaMacchia, B.; Longa, P.; et al. Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process. 2017. Available online: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip (accessed on 18 February 2020).
25. Hofheinz, D.; Hövelmanns, K.; Kiltz, E. A modular analysis of the Fujisaki-Okamoto transformation. In Proceedings of the Theory of Cryptography—15th International Conference, Baltimore, MD, USA, 12–15 November 2017; pp. 341–371.
26. Galbraith, S.D.; Petit, C.; Shani, B.; Ti, Y.B. On the Security of Supersingular Isogeny Cryptosystems. In Proceedings of the Advances in Cryptology—ASIACRYPT 2016—22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016; pp. 63–91.
27. The National Institute of Standards and Technology (NIST). Post-Quantum Cryptography Standardization, 2017–2018. Available online: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization (accessed on 18 February 2020).
28. Azarderakhsh, R.; Campagna, M.; Costello, C.; Feo, L.D.; Hess, B.; Jalali, A.; Jao, D.; Koziel, B.; LaMacchia, B.; Longa, P.; et al. Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process, round 2. 2019. Available online: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions/SIKE.zip (accessed on 18 February 2020).

29. Costello, C.; Longa, P.; Naehrig, M. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Proceedings of the Advances in Cryptology—ASIACRYPT 2016—22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016*; Robshaw, M.; Katz, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9814, pp. 572–601.
30. Montgomery, P.L. Modular multiplication without trial division. *Math. Comput.* **1985**, *44*, 519–521. [CrossRef]
31. Liu, Z.; Longa, P.; Pereira, G.; Reparaz, O.; Seo, H. FourQ on embedded devices with strong countermeasures against side-channel attacks. *IEEE Trans. Dependable Secure Comput.* **2018**. [CrossRef]
32. Banik, S.; Bogdanov, A.; Regazzoni, F. Exploring energy efficiency of lightweight block ciphers. In Proceedigs of the International Conference on Selected Areas in Cryptography, Sackville, Canada, 12–14 August 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 178–194.
33. Banik, S.; Bogdanov, A.; Regazzoni, F.; Isobe, T.; Hiwatari, H.; Akishita, T. Round gating for low energy block ciphers. In Proceedigs of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 3–5 May 2016. IEEE: Hoboken, NJ, USA, 2016; pp. 55–60.