SIKE Round 2 Speed Record on ARM Cortex-M4

Hwajeong Seo[1]*, Amir Jalali[2], and Reza Azarderakhsh[2,3]

[1]IT Department, Hansung University, Seoul, South Korea, hwajeong84@gmail.com
[2]Department of Computer and Electrical Engineering and Computer Science,
Florida Atlantic University, FL, USA,
{ajalali2016,razarderakhsh}@fau.edu
[3]PQSecure Technologies, LLC

Abstract. We present the first practical software implementation of Supersingular Isogeny Key Encapsulation (SIKE) round 2, targeting NIST's 1, 2, and 5 security levels on 32-bit ARM Cortex-M4 microcontrollers. The proposed library introduces a new speed record of SIKE protocol on the target platform. We achieved this record by adopting several state-of-the-art engineering techniques as well as highly-optimized hand-crafted assembly implementation of finite field arithmetic. In particular, we carefully redesign the previous optimized implementations of filed arithmetic on 32-bit ARM Cortex-M4 platform and propose a set of novel techniques which are explicitly suitable for SIKE/SIDH primes. Moreover, the proposed arithmetic implementations are fully scalable to larger bit-length integers and can be adopted over different security levels. The benchmark result on STM32F4 Discovery board equipped with 32-bit ARM Cortex-M4 microcontrollers shows that the entire key encapsulation over p434 takes about 326 million clock cycles (i.e. 1.94 seconds @168MHz). In contrast to the previous optimized implementation of the isogeny-based key exchange on low-power 32-bit ARM Cortex-M4, our performance evaluation shows feasibility of using SIKE mechanism on the target platform. In comparison to the most of the post-quantum candidates, SIKE requires an excessive number of arithmetic operations, resulting in significantly slower timings. However, its small key size makes this scheme as a promising candidate on low-end microcontrollers in the quantum era by ensuring the lower energy consumption for key transmission than other schemes.

Keywords: Post-quantum cryptography, SIDH, SIKE, Montgomery multiplication, ARM Cortex-M4

1 Introduction

The hard problems of traditional PKC (e.g. RSA and ECC) can be easily solved by using Shor's algorithm [26] and its variant on a quantum computer. The traditional PKC approaches cannot be secure anymore against quantum attacks.

* Corresponding Author

A number of post-quantum cryptography algorithms have been proposed in order to resolve this problem. Among them, Supersingular Isogeny Diffie-Hellman key exchange (SIDH) protocol proposed by Jao and De Feo is considered as a premier candidate for post-quantum cryptosystems [17]. Its security is believed to be secure even for quantum computers. SIDH is the basis of the Supersingular Isogeny Key Encapsulation (SIKE) protocol [2], which is currently under consideration by the National Institute of Standards and Technology (NIST) for inclusion in a future standard for post-quantum cryptography [27]. One of the attractive features of SIDH and SIKE is their relatively small public keys which are, to date, the most compact ones among well-established quantum-resistant algorithms. In spite of this prominent advantage, the "slow" speed of these protocols has been a sticking point which hinders them from acting like the post-quantum cryptography. Therefore, speeding up SIDH and SIKE has become a critical issue as it judges the practicality of these isogeny-based cryptographic schemes. In CANS'16, Koziel et al. presented first SIDH implementations on 32-bit ARM Cortex-A processors [22]. In 2017, Jalali et al. presented first SIDH implementations on 64-bit ARM Cortex-A processors [16]. In CHES'18, Seo et al. improved previous SIDH and SIKE implementations on high-end 32/64-bit ARM Cortex-A processors [25]. At the same time, the implementations of SIDH on Intel and FPGA are also successfully evaluated [10, 3, 19, 21]. Afterward, in 2018, first implementation of SIDH on low-end 32-bit ARM Cortex-M4 microcontroller was suggested [20]. The paper shows that an ephemeral key exchange (i.e. SIDHp751) on a 32-bit ARM Cortex-M4@120MHz requires 18.833 seconds to perform - too slow to use on low-end microcontrollers.

In this work, we challenge to the practicality of SIKE round 2 protocols for NIST PQC competition (i.e. SIKEp434, SIKEp503, and SIKEp751) on low-end microcontrollers. We present new optimized implementation of modular arithmetic for the case of low-end 32-bit ARM Cortex-M4 microcontroller. The proposed modular arithmetic, which is implemented on top of the SIKE round 2 reference implementation [1], demonstrates that the supersingular isogeny-based protocols are practical on 32-bit ARM Cortex-M4 microcontrollers.

2 Optimized SIKE/SIDH Arithmetic on ARM Cortex-M4

2.1 Multiprecision Multiplication

In this work, we describe the multi-precision multiplication method in multiplication structure and rhombus form.

Figure 1, 2, and 3 illustrate different strategies for implementing 256-bit multiplication on 32-bit ARM Cortex-M4 microcontroller. Let A and B be operands of length m bits each. Each operand is written as $A = (A[n-1], ..., A[1], A[0])$ and $B = (B[n-1], ..., B[1], B[0])$, where $n = \lceil m/w \rceil$ is the number of words to represent operands, and w is the computer word size (i.e. 32-bit). The result $C = A \cdot B$ is represented as $C = (C[2n-1], ..., C[1], C[0])$. In the rhombus form,

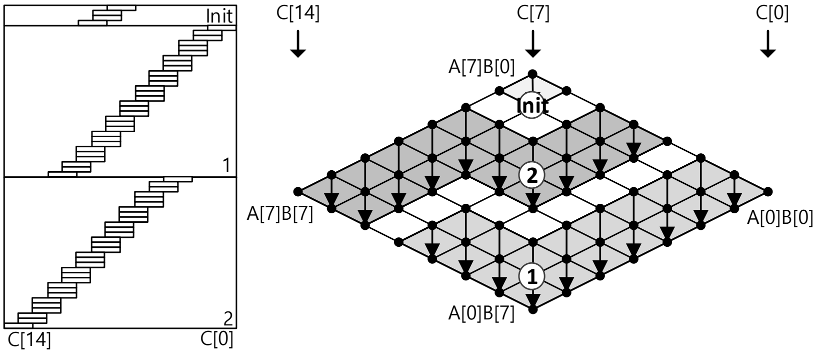

Fig. 1: 256-bit Operand Caching multiplication at the word-level where e is 3 on ARM Cortex-M4 [11], Init: initial block; ① → ②: order of rows.

the lowest indices ($i,\ j = 0$) of the product appear at the rightmost corner, whereas the highest indices ($i,\ j = n-1$) appear at the leftmost corner. A black arrow over a point indicates the processing of a partial product. The lowermost points represent the results $C[i]$ from the rightmost corner ($i = 0$) to the leftmost corner ($i = 2n-1$).

There are several works in the literature that studied the use of `UMAAL` instructions to implement multi-precision multiplication or modular multiplication on 32-bit ARM Cortex-M4 microcontrollers [8, 9, 11, 23, 20, 13]. Among them, Fujii et al. [11], Haase et al. [13], and Koppermann et al. [20] provided the most relevant optimized implementations to this work, targeting Curve25519 and SIDHp751 by using optimal modular multiplication and squaring methods.

In [11], authors combine the `UMAAL` instruction with (Consecutive) Operand Caching (OC) method for Curve25519 (i.e. 256-bit multiplication). The `UMAAL` instruction handles the carry propagation without additional costs in Multiplication ACcumulation (MAC) routine. The detailed descriptions are given in Figure 1. The size of operand caching is 3, which needs three rows ($3 = \lceil 8/3 \rceil$) for 256-bit multiplication on 32-bit ARM Cortex-M4. The multiplication starts from initial block and performs rows 1 and 2, sequentially. The inner loop follows column-wise (i.e. Product-Scanning) multiplication.

In [13], a highly-optimized usage of registers and the partial products are performed with the Operand Scanning (OS) method, targeting Curve25519 (i.e. 256-bit multiplication). The detailed descriptions are given in Figure 2. In particular, the order of partial products has an irregular pattern which only works for the target operand length (i.e. 256-bit multiplication) due to the extremely compact utilization of available registers in each partial product. However, for a larger length integer multiplication, this greedy approach is not suitable since the number of register is not enough to cache sufficient operands and intermediate results to achieve the optimal performance.

In [20], authors proposed an implementation of 1-level additive Karatsuba multiplication with Comba method (i.e. Product Scanning) as the underlying

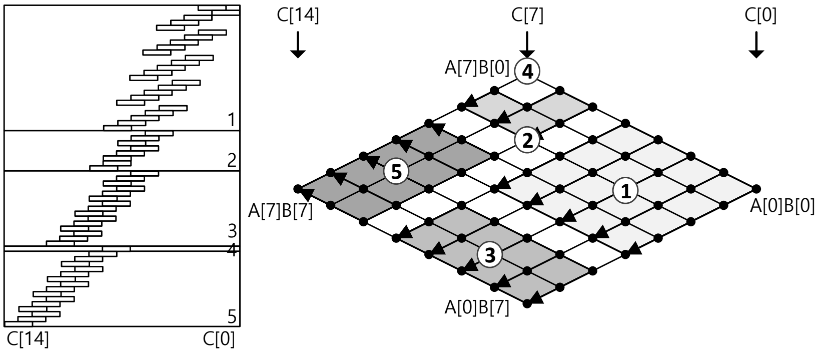

Fig. 2: 256-bit Operand Scanning multiplication at the word-level on ARM Cortex-M4 [13], ① → ② → ③ → ④ → ⑤ : order of rows.

multiplication strategy, targeting 768-bit multiplication . They integrated their arithmetic library into SIDHp751 and reported the first optimized implementation of SIDH on ARM Cortex-M4 microcontrollers. However, the product scanning is inefficient with the UMAAL instruction, since all the intermediate results for long integer multiplication cannot be stored into the small number of available registers. In order to improve their results, we studied the performance evaluation of 448/512/768-bit multiplication by replacing the Comba method with OC method, using the 1-level additive/subtractive Karatsuba multiplication. However, we realized that the Karatsuba approach is slower than original OC method with UMAAL instruction for large integer multiplication on Cortex-M4, due to the excessive number of number of addition, subtraction, bit-wise exclusive-or, and loading/storing intermediate results inside Karatsuba method. Furthermore, 32-bit ARM Cortex-M4 microcontroller provides same latency (i.e. 1 clock cycle) for both 32-bit wise unsigned multiplication with double accumulation (i.e. UMAAL) and 32-bit wise unsigned addition (i.e. ADD).

We acknowledge that on low-end devices, such as 8-bit AVR microcontrollers, Karatsuba method is one of the most efficient approaches for multi-precision multiplication. In these platforms, the MAC routine requires at least 5 clock cycles [14]. This significant overhead is efficiently replaced with relatively cheaper 8-bit addition/subtraction operation (i.e. 1 clock cycle). However, UMAAL instruction in ARM Cortex-M4 microcontroller can perform the MAC routine within 1 clock cycle. For this reason, it is hard to find a reasonable trade-off between MAC (i.e. 1 clock cycle) and addition/subtraction (i.e. 1 clock cycle) on the ARM Cortex-M4 microcontroller. Following the above analysis, we adopted the OC method for implementing multiplication in our proposed implementation. Moreover, in order to achieve the most efficient implementation of SIKE protocol on ARM Cortex-M4, we proposed three distinguished improvements to the original method which result in significant performance improvement compared to previous works. We describe these techniques in the following.

Table 1: Comparison of multiplication methods, in terms of memory-access complexity. The parameter d defines the number of rows within a processed block.

Method	Load	Store
Operand Scanning	$2n^2 + n$	$n^2 + n$
Product Scanning [5]	$2n^2$	$2n$
Hybrid Scanning [12]	$2\lceil n^2/d \rceil$	$2n$
Operand Caching [15]	$2\lceil n^2/e \rceil$	$\lceil n^2/e \rceil + n$
Refined Operand Caching (This work)	$2\lceil n^2/(e+1) \rceil + 3(\lfloor n/(e+1) \rfloor)$	$\lceil n^2/(e+1) \rceil + n$

Table 2: Comparison of multiplication methods for different Integer sizes, in terms of the number of memory access on 32-bit ARM Cortex-M4 microcontroller. The parameters d and e are set to 2 and 3, respectively.

Method	448-bit			512-bit			768-bit		
	Load	Store	Total	Load	Store	Total	Load	Store	Total
OS	406	210	616	528	272	800	1,176	600	1,776
PS	392	28	420	512	32	544	1,152	48	1,200
HS	196	28	224	256	32	288	576	48	624
OC	132	80	212	172	102	274	384	216	600
R-OC	107	63	170	140	80	220	306	168	474

Efficient register utilization The OC method follows the product-scanning approach for inner loop but it divides the calculation (i.e. outer loop) into several rows [15]. The number of rows directly affects the overall performance, since the OC method requires to load the operands and load/store the intermediate results by the number of rows[1]. Table 1 presents the comparison of memory access complexity depending on the multiplication techniques. Our optimized implementation (i.e. Refined Operand Caching) is based on the original OC method but we optimized the available registers and increased the operand caching size from e to $e + 1$. In the equation, the number of memory load by $3(\lfloor n/(e + 1) \rfloor)$ indicates the operand pointer access in each row.

Moreover, larger bit-length multiplication requires more memory access operations. Table 2 presents the number of memory access operations in OC method for different multi-precision multiplication size. In this table, our proposed R-OC method requires the least number memory access for different length multiplication. In particular, in comparison with original OC implementation, our proposed implementation reduces the total number of memory accesses by 19.8 %, 19.7 %, and 21 % for 448-bit, 512-bit, and 768-bit, respectively[2].

In order to increase the size of operand caching (i.e. e) by 1, we need at least 3 more registers to retain two 32-bit operand limbs and one 32-bit interme-

[1] The number of rows is $r = \lfloor n/e \rfloor$, where the number of needed words ($n = \lceil m/w \rceil$), the word size of the processor (w) (i.e. 32-bit), the bit-length of operand (m), and operand caching size (e) are given.

[2] Compared with original OC implementation, we reduce the number of row by 1 ($4 \rightarrow 3$), 2 ($5 \rightarrow 3$), and 2 ($7 \rightarrow 5$) for 448-bit, 512-bit, and 768-bit, respectively.

Table 3: Comparison of register utilization of the proposed method with previous works.

Registers	Fujii et al. [11]	Haase et al. [13]	This work
`R0`	Result pointer	Temporal pointer	Temporal pointer
`R1`	Operand A pointer	Operand A #1	Temporal register #1
`R2`	Operand B pointer	Operand B #1	Operand A #1
`R3`	Result #1	Operand B #2	Operand A #2
`R4`	Result #2	Operand B #3	Operand A #3
`R5`	Result #3	Operand B #4	Operand A #4
`R6`	Operand A #1	Operand B #5	Operand B #1
`R7`	Operand A #2	Result #1	Operand B #2
`R8`	Operand A #3	Result #2	Operand B #3
`R9`	Operand B #1	Result #3	Operand B #4
`R10`	Operand B #2	Result #4	Result #1
`R11`	Operand B #3	Result #5	Result #2
`R12`	Temporal register #1	Temporal register #1	Result #3
`R13; SP`	Stack pointer	Stack pointer	Stack pointer
`R14; LR`	Temporal register #2	Temporal register #2	Result #4
`R15; PC`	Program counter	Program counter	Program counter

diate result value. To this end, we redefine the register assignments inside our implementation. We saved one register for the result pointer by storing the intermediate results into stack. Moreover, we observed that in the OC method, both operand pointers are not used at the same time in the row. Therefore, we don't need to maintain both operand pointers in the registers during the computations. Instead, we store them to the stack and load one by one on demand.

Using the above techniques, we saved three available registers and utilized them to increase the size of operand caching by 1. In particular, three registers are used for operand A, operand B, and intermediate result, respectively. We state that our utilization technique imposes an overhead in memory access for operand pointers. However, since in each row, only three memory accesses are required, the overall overhead is negligible to the obtained performance benefit. We provide a detailed comparison of register assignments of this work with previous implementations in Table 3.

Optimized front parts As it is illustrated in Figure 3, our R-OC method starts from an initialization block (`Init` section). In the `Init` section, both operands are loaded from memory to registers and the partial products are computed. From the `row1`, only one operand pointer is required in each column. The front part (i.e. `I-F` and `1-F`) requires partial products by increasing the length of column to 4.

Fujii et al. [11] implemented the front parts using carry-less MAC routines. In their approach, they initialized up to two registers to store the intermediate results in each column. Figure 4 illustrates their approach. Since the `UMLAL`

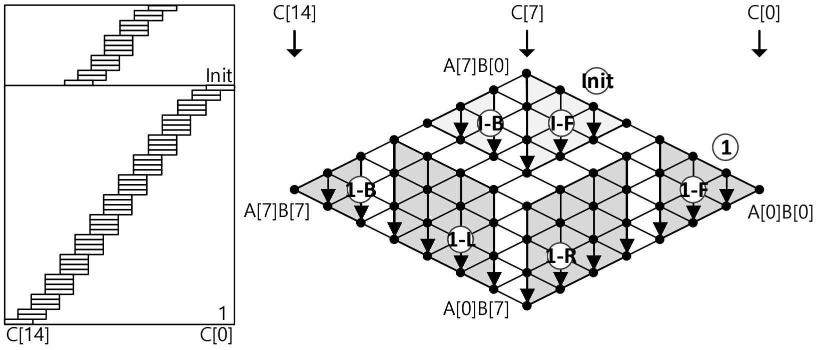

Fig. 3: Proposed 256-bit Refined Operand Caching multiplication at the word-level where e is 4 on ARM Cortex-M4, Ⓘnit: initial block; ①: order of rows; Ⓕ: front part; Ⓡ: middle right part; Ⓛ: middle left part; Ⓑ: back part.

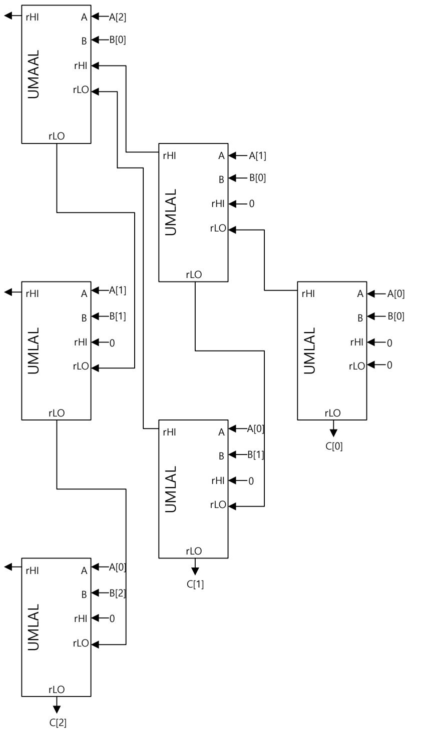

Fig. 4: 3-word integers with the product scanning approach using the `UMLAL` and `UMAAL` instructions for front part of OC method [11].

and `UMAAL` instructions need to update current values inside the registers, the initialized registers are required.

In order to optimize the explicit register initialization, we redesign the front part with product scanning. In contrast to Fujii's approach, we used `UMULL` and `UMAAL` instructions. As a result, the register initialization is performed together with unsigned multiplication (i.e. `UMULL`). This technique improves the overall clock cycles since each instruction directly assigns the results to the target registers. In particular, we are able to remove all the register initialization routines, which is 9 clock cycles for each front part compared to [11]. Moreover, the intermediate results are efficiently handled with carry-less MAC routines by using the `UMAAL` instructions. Figure 5 presents our 4-word strategy in further details.

Efficient instruction ordering The ARM Cortex-M4 microcontrollers are equipped with 3-stage pipeline in which the instruction fetch, decode, and execution are performed in order. As a result, any data dependency between consecutive instructions imposes pipeline stalls and degrades the overall performance considerably. In addition to the previous optimizations, we reordered the MAC routine instructions in a way which removes data dependency between instructions, resulting in minimum pipeline stalls. The proposed approach is presented in Figure 5 (`1-R` section). In this Figure, the operand and intermediate result are loaded from memory and partial products are performed column-wise as follows:

```
  ⋮
LDR   R6,[R0,#4*4] //Loading operand B[4] from memory
LDR   R1,[SP,#4*4] //Loading intermediate result C[4] from memory
UMAAL R14,R10,R5,R7 //Partial product (B[1]*A[3])
UMAAL R14,R11,R4,R8 //Partial product (B[2]*A[2])
UMAAL R14,R12,R3,R9 //Partial product (B[3]*A[1])
UMAAL R1,R14,R2,R6  //Partial product (B[4]*A[0])
  ⋮
```

The intermediate result ($C[4]$) is loaded to the `R1` register. At this point, updating `R1` register in the next instruction results in pipeline stall. To avoid this situation, first, we updated the intermediate results into other registers (`R10, R11, R12, R14`), while `R1` register was updated during the last step of MAC. We followed a similar approach in `1-L` section, where operand (A) pointer is loaded to a temporary register, and then the column-wise multiplications are performed with the operands ($A[4]$, $A[5]$, $A[6]$, and $A[7]$). In the back part (i.e. `1-B`), the remaining partial products are performed without operand loading. This is efficiently performed without carry propagation by using the `UMAAL` instructions.

To compare the efficiency of our proposed techniques with previous works, we evaluated the performance of our 256-bit multiplication with the most relevant works on Cortex-M4 platform. To obtain a fair and uniform comparison, we benchmarked the proposed implementations in [11, 13][3,4] with our implementation on our development environment.

[3] Fujii et al. `https://github.com/hayatofujii/curve25519-cortex-m4`

[4] Haase et al. `https://github.com/BjoernMHaase/fe25519`

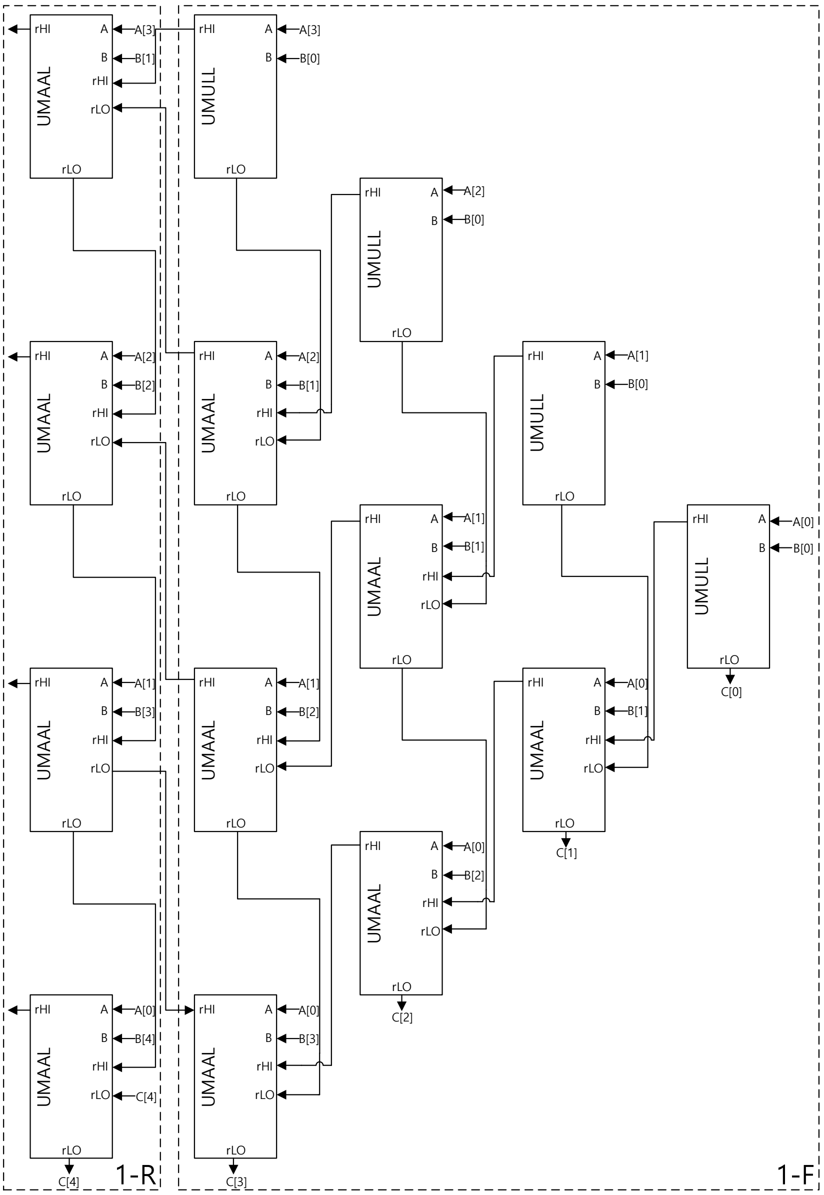

Fig. 5: 4-word integers with the product scanning approach using the UMULL and UMAAL instructions for front part of OC method.

Table 4: Comparison results of 256-bit multiplication on ARM Cortex-M4 microcontrollers.

Methods	Timings $[cc]$	Scalability	Bit length
Fujii et al. [11]	239	✓	256
Haase et al. [13]	212	✗	256
This work	196	✓	256

Table 4 presents the performance comparison of our library with previous works in terms of clock cycles. We observe that our proposed multiplication implementation method is faster than previous optimized implementation on the same platform. Furthermore, in contrast to the compact implementation of 256-bit multiplication in [13], our approach provides scalability to larger integer multiplication without any significant overhead.

2.2 Multiprecision Squaring

Most of the optimized implementations of cryptography libraries use optimized multiplication for computing the square of an element. However, squaring can be implemented more efficiently since using one operand reduces the overall number of memory accesses by half, while many redundant partial products can be removed (i.e. $A[i] \times A[j] + A[j] \times A[i] = 2 \times A[i] \times A[j]$).

Similar to multiplication, squaring implementation consists of partial products of the input operand limbs. These products can be divided into two parts: the products which have two operands with the same value and the ones in which two different values are multiplied. Computing the first group is straightforward and it is only computed once for each limb of operand. However, computing the latter products with different values and doubling the result can be performed in two different ways: doubled-result and doubled-operand. In doubled-result technique, partial products are computed first and the result is doubled afterwards ($A[i] \times A[j] \rightarrow 2 \times A[i] \times A[j]$), while in doubled-operand, one of the operands is doubled and then multiplied to the other value ($2 \times A[i] \rightarrow 2 \times A[i] \times A[j]$).

In the previous works [11, 13], authors adopted the doubled-result technique inside squaring implementation. Figure 6 and 7 show their techniques for implementing optimized squaring on Cortex-M4 platform. The red parts in the figures present the partial products where the input values are the same and the black dots with gray background represent the doubled-result products.

Figure 6 demonstrates Sliding Block Doubling (SBD) based squaring method in [11]. This method is based on the product scanning approach. The squaring consists of two routines: initialization and row 1 computation. The intermediate results are doubled column-wise as the row 1 computations are performed.

Figure 7 presents the Operand Scanning (OS) based squaring method in [13]. In contrast to previous method, computations are performed row-wise. However, the intermediate results are doubled in each column. Note that in this method, the order of computation is designed explicitly for 256-bit operand to maximize

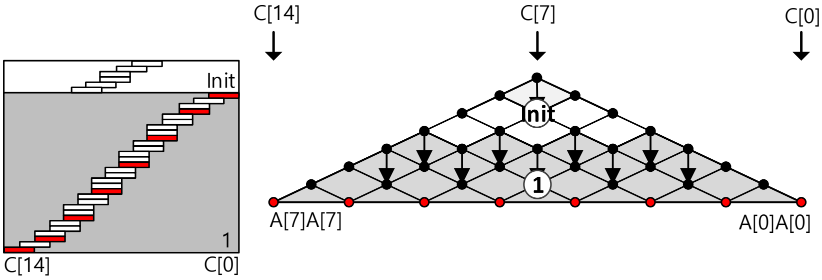

Fig. 6: 256-bit Sliding Block Doubling squaring at the word-level on ARM Cortex-M4, Init: initial block; ①: order of rows [11].

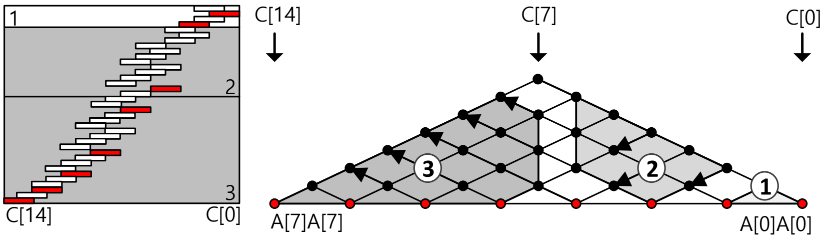

Fig. 7: 256-bit Operand Scanning squaring at the word-level on ARM Cortex-M4, ① → ② → ③: order of rows [13].

the operand caching. Similar to their multiplication implementation, the proposed method does not provide scalability to larger bit-length multiplications.

In this work, we proposed a hybrid approach for implementing a highly-optimized squaring operation which is explicitly suitable for SIKE/SIDH application. In general, doubling operation may result in one bit overflow which requires an extra word to retain. However, in the SIDH/SIKE settings, moduli are smaller than multiple of 32-bit word (434-bit, 503-bit, and 751-bit) which provide an advantage for optimized arithmetic design. Taking advantage of this fact, we designed our squaring implementation based on doubled-operand approach. We divided our implementation into three parts: one sub-multiplication and two sub-squaring operations. We used R-OC for sub-multiplication and SBD for sub-squaring operations. Figure 8 illustrates our hybrid method in detail. First, the input operand is doubled and stored into the stack memory. Taking advantage of doubled-operand technique, we perform the initialization part by using R-OC method.

Second, the remaining rows 1 and 2 are computed based on SBD methods. In contrast to previous SBD method, all the doubling operations on intermediate results are removed during MAC routines. This saves several registers to double the intermediate results since doubled-results have been already computed. Furthermore, our proposed method is fully scalable and can be simply adopted to larger integer squaring.

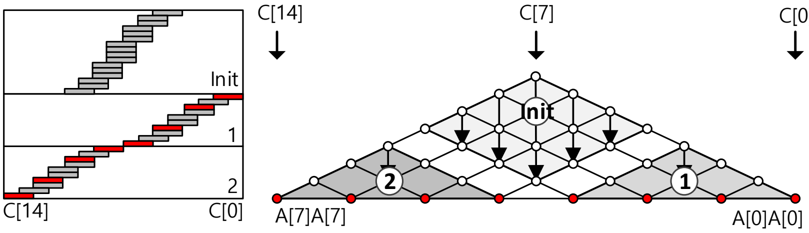

Fig. 8: 255-bit proposed squaring at the word-level on ARM Cortex-M4, Init: initial block; ① → ②: order of rows.

Table 5: Comparison results of 255/256-bit squaring on ARM Cortex-M4 microcontrollers.

Methods	Timings $[cc]$	Scalability	Bit length
Fujii et al. [11]	218	✓	256
Haase et al. [13]	141	✗	256
This work	136	✓	255

In order to verify the performance improvement of our proposed approach, we benchmarked our 255-bit squaring implementation with the most optimized available implementations in the literature. Table 5 presents the performance comparison of our method with previous implementations on our target platform.

Our hybrid method outperforms previous implementations of 256-bit squaring, while in contrast to [13], it is scalable to larger parameter sets. In particular, it enabled us to implement the same strategy for computing SIKE/SIDH arithmetic over larger finite fields.

2.3 Modular Reduction

Modular multiplication is a performance-critical building block in SIDH and SIKE protocols. One of the most well-known techniques used for its implementation is Montgomery reduction [24]. We adapt the implementation techniques described in sections 2.1 and 2.2 to implement modular multiplication and squaring operations. Specifically, we target the parameter sets based on the primes p434, p503, and p751 for SIKE round 2 protocol [6, 1]. Montgomery multiplication can be efficiently exploited and further simplified by taking advantage of so-called "Montgomery-friendly" modulus, which admits efficient computations, such as *all-zero* words for lower part of the modulus.

The efficient optimizations for the modulus were first pointed out by Costello et al. [6] in the setting of SIDH when using modulus of the form $2^x \cdot 3^y - 1$ (referred to as "SIDH-friendly" primes) are exploited by the SIDH library [7].

In CHES'18, Seo et al. suggested the variant of Hybrid-Scanning (HS) for "SIDH-friendly" Montgomery reduction on ARM Cortex-A15 [25]. Similar to OC method, the HS method also changes the operand pointer when the row is

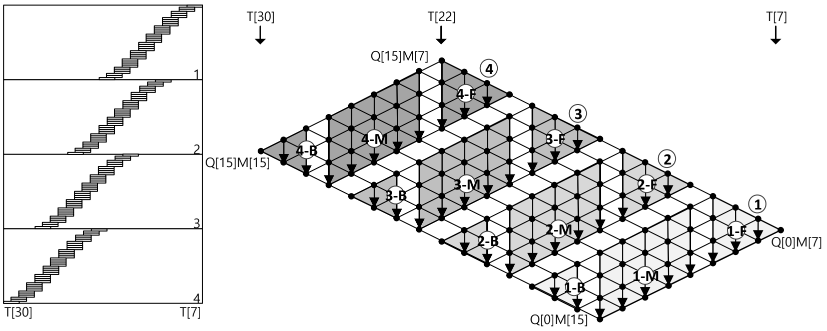

Fig. 9: 503-bit “SIDH-friendly” Montgomery reduction at the word-level, where d is 4 on ARM Cortex-M4, ① → ② → ③ → ④: order of rows; Ⓕ: front part; Ⓜ: middle part; Ⓑ: back part; where M, R, T, and Q are modulus, Montgomery radix, intermediate results, and quotient ($Q \leftarrow T \cdot M' \bmod R$).

changed. By using the register utilization described in Section 2.1, we increase the parameter d by 1 (3 → 4. Moreover, the initial block is also optimized to avoid explicit register initialization and the MAC routine is implemented in the pipeline-friendly approach. Compared with integer multiplication, the Montgomery reduction requires fewer number of registers to be reserved. Since the intermediate result pointer and operand Q pointer are identical value (i.e. stack), we only need to maintain one address pointer to access both values. Furthermore, the modulus for SIKE (i.e. operand M; SIKEp434, SIKEp503, and SIKEp751) is a static value. As a result, instead of obtaining values from memory, we assign the direct values to the registers. This step can be performed with the two instructions, such as MOVW and MOVT. The detailed 32-bit value assignment (e.g. 0x87654321) to register R1 is given as follows:

```
  ⋮
MOVW R1, #0x4321 //R1 = #0x4321
MOVT R1, #0x8765 //R1 = #0x8765 ≪ 16 | R1
  ⋮
```

In Figure 9, the 503-bit “SIDH-friendly” Montgomery reduction on ARM Cortex-M4 microcontroller is described. The Montgomery reduction starts from row 1, 2, 3, to 4.

In the front of row 1 (i.e. 1-F), the operand Q is loaded from memory and the operand M is directly assigned using constant value. The multiplication accumulates the intermediate results from memory using the operand Q pointer and stored them into the same memory address. In the middle of row 1 (i.e. 1-M), the operand Q is loaded and the intermediate results are also loaded and stored, sequentially. In the back of row 1 (i.e. 1-B), the remaining partial products are computed. Furthermore, the intermediate carry values are stored into stack and used in the following rows.

Table 6: Comparison results of modular multiplication and squaring for SIDH on 32-bit ARM Cortex-M4 microcontrollers.

Methods	Timings [cc]			Modulus	Processor
	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ sqr	reduction		
This work	1,110	981	544	$2^{216} \cdot 3^{137} - 1$	ARM Cortex-M4
SIDH v3.0 [7]	25,399	–	10,917	$2^{250} \cdot 3^{159} - 1$	ARM Cortex-M4
This work	1,333	1,139	654		
Bos et al. [4]	–	–	3,738	$2^{372} \cdot 3^{239} - 1$	ARM Cortex-A8
SIDH v3.0 [7]	55,178	–	23,484		ARM Cortex-M4
Koppermann et al. [20]	7,573	–	3,254		
This work	2,744	2,242	1,188		

Using the above techniques, we are able to reduce the number of row by 1 (5 → 4), 2 (6 → 4), and 2 (8 → 6) for 448-bit, 512-bit, and 768-bit, respectively, compared to original implementation of HS based Montgomery reduction.

Recently, Bos et al. [4] and Koppermann et al. [20] proposed highly optimized techniques for implementation of modular multiplication. They utilized the product-scanning methods for modular reduction. However, our proposed method outperfoms both implementations in terms of clock cycles. In particular, our proposed method provides more than 2 times faster result compared to Bos et al. [4], while the benchmark results in [4] were obtained on the high-end ARMv7 Cortex-A8 processors which is equipped with 15 pipeline stages and is dual-issue super-scalar. Table 6 shows the detailed performance comparison of multiplication, squaring, and reduction over SIDH/SIKE primes in terms of clock cycles. We state that, the benchmark results for [7] are based on optimized C implementation and they are presented solely as a comparison reference between portable and target-specific implementations.

2.4 Modular Addition and Subtraction

Modular addition operation is performed as a long integer addition operation followed by a subtraction from the prime. To have a fully constant-time arithmetic implementation, the final reduction is performed using a masked bit. In this case, even if the addition result is inside the field, a redundant subtraction is performed, so the secret values cannot be retrieved using power and timing attacks. The detailed operations are presented in the following:

- `Modular addition: (A+B) mod P`
  ① `C←A+B` ② `{M,C}←C-P` ③ `C←C+(P&M).`
- `Modular subtraction: (A-B) mod P`
  ① `{M,C}←A-B` ② `C←C+(P&M).`

Previous optimized implementations of modular addition on Cortex-M4 [25, 20], provided the simple masked technique using hand-crafted assembly. However, In this work, we optimized this approach further by introducing three techniques:

- Proposed modular addition: (A+B) mod P
  ① {M,C}←A+B-P ② C←C+(P&M).

First, we take advantage of the special shape of SIDH-friendly primes which have multiple words equal to 0xFFFFFFFF. Since this value is the same for multiple limbs, we load it once inside a register and use it for multiple single-precision subtraction. This operand re-using technique reduces the number of memory access by n and $\frac{n}{2}$ for modular addition and modular subtraction, where the number of needed words ($n = \lceil m/w \rceil$), the word size of the processor (w) (i.e. 32-bit), and the bit-length of operand (m) are given, respectively.

Second, we combine Step ① (addition) and ② (subtraction) into one operation ({M,C}←A+B-P). In order to combine both steps, we catch both intermediate carry and borrow, while we perform the combined addition and subtraction operation.

Figure 10 illustrates the proposed technique in details. In this Figure, first, 4-word addition operations ($A[0 \sim 3] + B[0 \sim 3]$) compute the addition result. Subsequently, a single register is set to constant (i.e. 0xFFFFFFFF), which is used for the carry catching step. In Figure 10, this step is shown in the last row of fourth column. When the carry overflow happens from fourth word addition (i.e. $A[3] + B[3] + CARRY$), the carry catcher register is set to $2^{32} - 1$ (i.e. 0xFFFFFFFF ← 0xFFFFFFFF + 0xFFFFFFFF + 0x00000001) by using the constant (i.e. 0xFFFFFFFF) in last row of fourth column (Constant + Constant + Carry). Otherwise, the carry catcher register is set to $2^{32} - 2$ (i.e. 0xFFFFFFFE ← 0xFFFFFFFF + 0xFFFFFFFF + 0x00000000).

This addition operation stores the carry bit to the first bit of carry catcher register. The carry value in carry catcher register is used for the following addition steps (second column in the Figure 10).

The stored carry in the first bit is shifted to the 32nd bit by using the barrel-shifter module. Afterward, the value is added to the constant (i.e. 0xFFFFFFFF). If the first bit of carry catcher is set, the carry happens (i.e. 0x00000001≪31 + 0xFFFFFFFF). Otherwise, no carry happens (i.e. 0x00000000≪31 + 0xFFFFFFFF).

Similarly, we obtained the borrow bit. The results of 4-word addition operations ($A[0 \sim 3] + B[0 \sim 3]$) are subtracted by modulus ($P[0 \sim 3]$) in the third column. When the borrow happens from fourth word subtraction (i.e. $A[3]+B[3]-P[3]-BORROW$), the borrow catcher register is set to $2^{32}-1$ (i.e. 0xFFFFFFFF ← 0x00000000 - 0x00000001) in last row of third column (Zero - Borrow). Otherwise, the borrow catcher register is set to 0 (i.e. 0x00000000 ← 0x00000000 - 0x00000000). The borrow bit in borrow catcher register is used for the following subtraction steps. To obtain the borrow bit, the zero constant is subtracted by the borrow catcher register. For one constant register optimization, we used the address pointer instead of zero constant.

Since the address pointer of 32-bit ARM Cortex-M4 microcontroller is aligned by 4-byte (i.e. 32-bit), the address is always ranging from 0 (i.e. 0x00000000) to $2^{32} - 4$ (0xFFFFFFFC). When the borrow catcher register is set, we can get the borrow bit through subtraction (e.g. Pointer - 0xFFFFFFFF where pointer is ranging from 0 to $2^{32} - 4$). Otherwise, no borrow happens. The combined

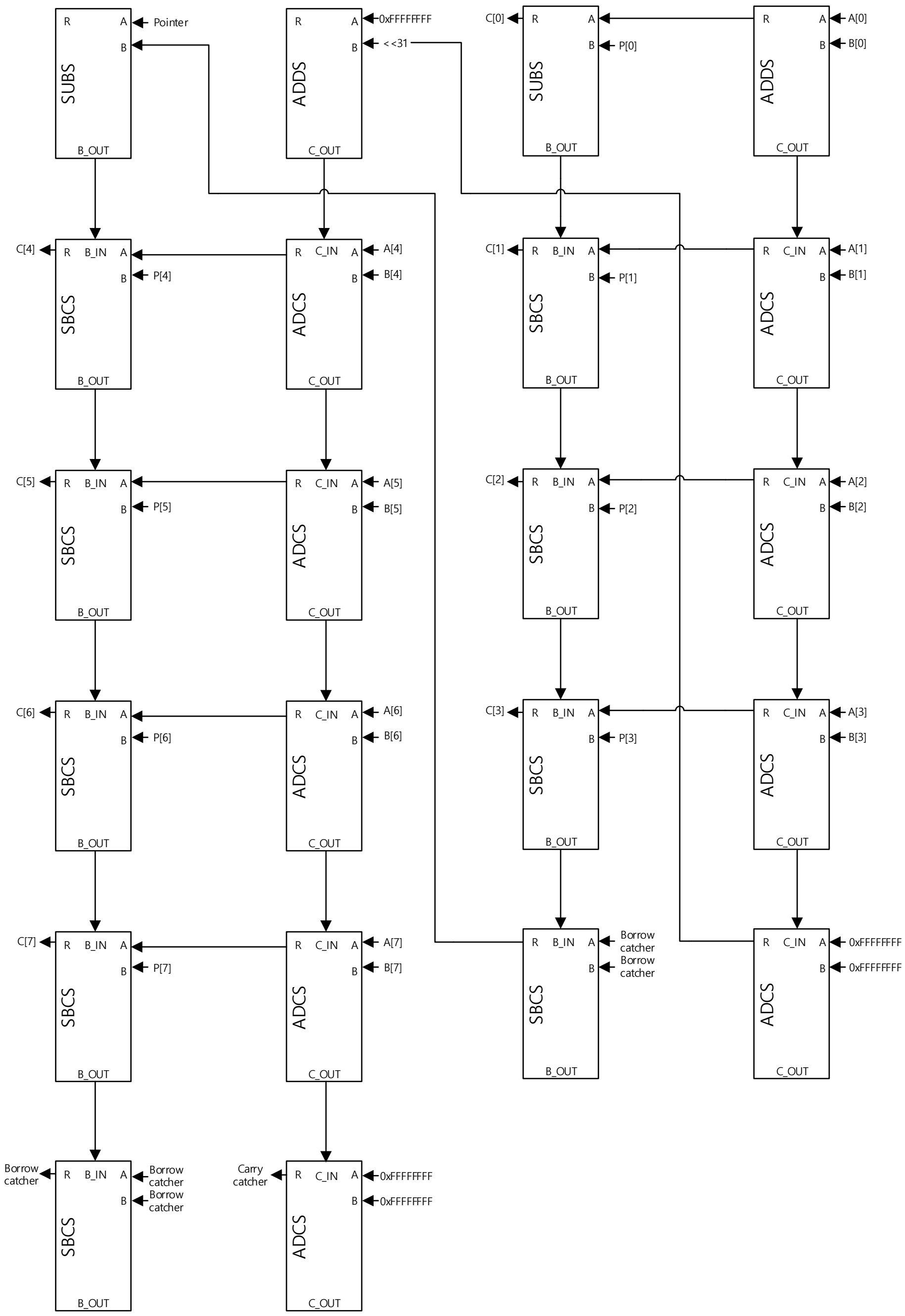

Fig. 10: Initial part of step ① in 512-bit modular addition on ARM Cortex-M4 (i.e. `A[0∼7]+B[0∼7]-P[0∼7]`).

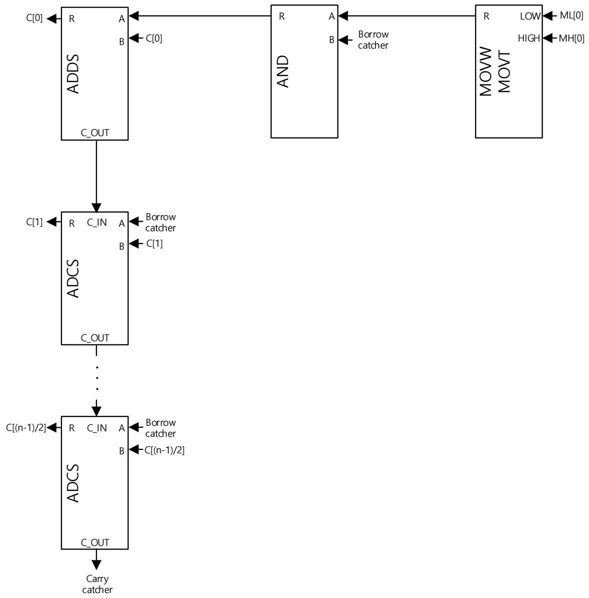

Fig. 11: Initial part of of step ② in 512-bit modular addition/subtraction on ARM Cortex-M4 (i.e. `C[0∼(n-1)/2]+(P[0∼(n-1)/2]&M)`).

modular addition routine reduces the number of memory access by $2n$ since we can avoid both loading and storing the intermediate results.

In addition to the above techniques, the masked addition routine is also optimized. This is shown as Step ② of modular addition and subtraction. When the mask value is set to `0xFFFFFFFF`, the lower part of SIDH modulus is also `0xFFFFFFFF`. Otherwise, both values are set to zero. We optimized the modulus setting (`MOVW/MOVT`) and masking operation (`AND`) for lower part of SIDH modulus. The detailed descriptions for initial part of step ② in 512-bit modular addition/subtraction are given in Figure 11.

Using the above optimization techniques, we are able to reduce the number of memory access for modular addition and subtraction by $3n$ ($9n \rightarrow 6n$) and $n/2$ ($6n \rightarrow 11n/2$), respectively.

We benchmarked the proposed optimized addition and subtraction implementations on our target platform. We provide the performance evaluation of this work and previous works over different security levels in Table 7. Compared to previous works, the proposed method improved the performance by 16.7 % and 14.7 % for modular addition and subtraction, respectively.

3 Performance Evaluation

In this section, we present the performance evaluation of our proposed SIDH/SIKE implementations on 32-bit ARM Cortex-M4 microcontrollers. We implemented

Table 7: Comparison results of modular addition and subtraction for SIDH/SIKE on ARM Cortex-M4 microcontrollers.

Methods	Timings [cc]		Modulus	Processor
	$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub		
This work	254	208	$2^{216} \cdot 3^{137} - 1$	ARM Cortex-M4
SIDH v3.0 [7]	1,078	740	$2^{250} \cdot 3^{159} - 1$	ARM Cortex-M4
Seo et al. [25]	326	236		
This work	275	223		
SIDH v3.0 [7]	1,579	1,092	$2^{372} \cdot 3^{239} - 1$	ARM Cortex-M4
Koppermann et al. [20]	559	419		
Seo et al. [25]	466	333		
This work	388	284		

highly-optimized arithmetic, targeting SIKE round 2 primes adapting our optimized techniques for multiplication, squaring, reduction, and addition/subtraction. We integrate our arithmetic libraries to the SIKE round 2 reference implementation [1] to evaluate the feasibility of adopting this scheme on low-end Cortex-M4 microcontrollers.

All the arithmetic is implemented in ARM assembly and the libraries are compiled with GCC with optimization flag set to `-O3`.[5]

Table 8 and 9 present the comparison of our proposed library with highly optimized implementations in the literature over different security levels. The optimized C implementation timings by Costello et al. [7] and the reference C implementation of SIKE [1] illustrate the importance of target-specific implementations of SIDH/SIKE low-end microcontrollers such as 32-bit ARM Cortex-M4. In particular, compared to optimized C Comba based implementation in SIDH v3.0, the proposed modular multiplication for 503-bit and 751-bit provide 19.05x and 20.10x improvement, respectively.

The significant achieved performance improvement in this work is the result of our highly-optimized arithmetic library. Specifically, our tailored multiplication minimizes pipeline stalls on ARM Cortex-M4 3-stage pipeline, resulting in remarkable timing improvement compared to previous works.

Moreover, the proposed implementation achieved 362 and 977 million clock cycles for total computation of SIDHp503 and SIDHp751, respectively. The results are improved by 10.51x and 12.97x for SIDHp503 and SIDHp751, respectively. In comparison with the most relevant work, our proposed modular multiplication and SIDHp751 outperforms the optimized implementation in [20] by 2.75x and 4.35x, respectively.

Compared with other NIST PQC round 2 schemes, the SIKE protocol shows slower execution time but the SIKE protocols show the most competitive memory

[5] Our library will be publicly available in the near future.

Table 8: Comparison of SIDHp434, SIDHp503, and SIDHp751 protocols on the ARM Cortex-M4 microcontrollers. Timings are reported in terms of clock cycles.

Implementation	Language	Timings [cc]				Timings [$cc \times 10^6$]				
		$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ sqr	Alice R1	Bob R1	Alice R2	Bob R2	Total
SIDHp434										
This work	ASM	254	208	1,110	981	65	74	54	62	255
SIDHp503										
SIDH v3.0 [7]	C	1,078	740	25,399	–	986	1,086	812	924	3,808
This work	ASM	275	223	1,333	1,139	95	104	76	87	362
SIDHp751										
SIDH v3.0 [7]	C	1,579	1,092	55,178	–	3,246	3,651	2,669	3,112	12,678
Koppermann et al. [20]	ASM	559	419	7,573	–	1,025	1,148	967	1,112	4,252
This work	ASM	388	284	2,744	2,242	252	284	205	236	977

Table 9: Comparison of NIST PQC round 2 protocols on the ARM Cortex-M4 microcontrollers. Timings are reported in terms of clock cycles. Koppermann et al. [20] does not provide results on SIKE implementations.

Implementation	Language	Timings [cc]				Timings [$cc \times 10^6$]				Memory [bytes]		
		$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ sqr	KeyGen	Encaps	Decaps	Total	KeyGen	Encaps	Decaps
SIKEp434												
This work	ASM	254	208	1,110	981	74	122	130	326	6,580	6,916	7,260
SIKEp503												
SIDH v3.0 [7]	C	1,078	740	25,399	–	1,086	1,799	1,912	4,797	–	–	–
This work	ASM	275	223	1,333	1,139	104	172	183	459	6,204	6,588	6,974
SIKEp751												
SIDH v3.0 [7]	C	1,579	1,092	55,178	–	3,651	5,918	6,359	15,928	–	–	–
This work	ASM	388	284	2,744	2,242	282	455	491	1,228	11,116	11,260	11,852
NIST PQC Round 2 [18]												
Frodo640-AES	ASM	–	–	–	–	42	46	47	135	31,116	51,444	61,820
Frodo640-CSHAKE	ASM	–	–	–	–	81	86	87	254	26,272	41,472	51,848
Kyber512	ASM	–	–	–	–	0.7	0.9	1.0	2.6	6,456	9,120	9,928
Kyber768	ASM	–	–	–	–	1.2	1.4	1.4	4.0	10,544	13,720	14,880
Kyber1024	ASM	–	–	–	–	1.7	2.1	2.1	5.9	15,664	19,352	20,864
Newhope1024CCA	ASM	–	–	–	–	1.2	1.9	1.9	5	11,152	17,448	19,648
Saber	ASM	–	–	–	–	0.9	1.2	1.2	3.3	12.616	14,896	15,992

utilization for encapsulation and decapsulation[6]. Furthermore, small key size of SIKE ensures the lower energy consumption for key transmission than other schemes. The low-energy consumption is the most critical requirement for low-end (battery-powered) microcontrollers.

In Table 10, we evaluated the practicality of SIDH protocols on both high-end ARM Cortex-A family of processors and low-end ARM Corex-M4 microcontrollers by measuring the timing in seconds.

The fastest implementations of SIDHp503 on 64-bit ARMv8 Cortex-A53 and Cortex-A72 only require 0.041 second and 0.021 second, respectively. For the case of 32-bit ARMv7 Cortex-A15, the SIDHp751 protocol is performed in 0.157 second. This results emphasize that SIDH protocol is already a practical solution for those "high-end" processors.

Finally, prior to this work, supersingular isogeny-based cryptography was assumed to be unsuitable to use on low-end devices due to the nonviable perfor-

[6] SIKEp434 requires more memory than SIKEp503 since SIKEp434 allocates more temporal storage than SIKEp503 in Fermat based inversion.

mance evaluations [20][7]. However, in contrast to benchmark results in [20], our SIKE and SIDH implementation for NIST's 1, 2, and 5 security levels are practical and can be used in real settings. The proposed implementation of SIDHp434 only requires 0.813 second, which shows that the quantum-resistant key exchange from isogeny of supersingular elliptic curve is a practical solution on low-power microcontrollers.

Table 10: Comparison of SIDH based key exchange protocols on high-end (ARM Cortex-A series) processors and low-end (ARM Cortex-M4) microcontrollers. Timings are reported in terms of seconds.

Protocol	Implementation	Platform	Freq	Latency [sec.]		Comm. [bytes]	
			[MHz]	Alice	Bob	A→B	B→A
High-end ARM Processors							
SIDHp503	[22]	32-bit ARMv7 Cortex-A8	1,000	0.216	0.229	378	378
	[22]	32-bit ARMv7 Cortex-A15	2,300	0.064	0.067	378	378
	[25]		2,000	0.042	0.046	378	378
	[2]	64-bit ARMv8 Cortex-A53	1,512	0.061	0.050	378	378
	[25]		1,512	0.050	0.041	378	378
	[2]	64-bit ARMv8 Cortex-A72	1.992	0.030	0.025	378	378
	[25]		1.992	0.025	0.021	378	378
SIDHp751	[22]	32-bit ARMv7 Cortex-A8	1,000	1.406	1.525	564	564
	[22]	32-bit ARMv7 Cortex-A15	2,300	0.340	0.368	564	564
	[25]		2,000	0.135	0.157	564	564
Low-end ARM Microcontrollers							
SIDHp434	This work	32-bit ARMv7 Cortex-M4	168	0.715	0.813	326	326
SIDHp503	This work		168	1.028	1.143	378	378
SIDHp751	[20]		120	16.590	18.833	564	564
	This work		168	2.727	3.099	564	564

4 Acknowledgement

The authors would like to thank the reviewers for their comments.

This work of Hwajeong Seo was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1C1B5075742) and partly supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2019-2014-1-00743) supervised by the IITP(Institute for Information communications Technology Planning Evaluation).

References

1. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev,

[7] Authors reported 18 seconds to key exchange on the ARM Cortex-M4 @120 MHz processor

and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, round 2, 2019. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions/SIKE.zip`.

2. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, 2017. Available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip`.
3. J. Bos and S. Friedberger. Arithmetic considerations for isogeny based cryptography. *IEEE Transactions on Computers*, 2018.
4. J. W. Bos and S. Friedberger. Faster modular arithmetic for isogeny based crypto on embedded devices. *IACR Cryptology ePrint Archive*, 2018:792, 2018.
5. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM systems journal*, 29(4):526–538, 1990.
6. C. Costello, P. Longa, and M. Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In M. Robshaw and J. Katz, editors, *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.
7. C. Costello, P. Longa, and M. Naehrig. SIDH Library. `https://github.com/Microsoft/PQCrypto-SIDH`, 2016–2018.
8. W. de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. PhD thesis, Ph. D. thesis, Eindhoven University of Technology (Sep 2015), 2015.
9. F. De Santis and G. Sigl. Towards side-channel protected X25519 on ARM Cortex-M4 processors. *Proceedings of Software performance enhancement for encryption and decryption, and benchmarking, Utrecht, The Netherlands*, pages 19–21, 2016.
10. A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, 2018.
11. H. Fujii and D. F. Aranha. Curve25519 for the Cortex-M4 and beyond. *Progress in Cryptology-LATINCRYPT*, 35:36–37, 2017.
12. N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *International workshop on cryptographic hardware and embedded systems*, pages 119–132. Springer, 2004.
13. B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–48, 2019.
14. M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. *Journal of Cryptographic Engineering*, 5(3):201–214, 2015.
15. M. Hutter and E. Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 459–474. Springer, 2011.
16. A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao. Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM. *IEEE Transactions on Dependable and Secure Computing*, 2017.
17. D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In B. Yang, editor, *Post-Quantum Cryptography (PQCrypto 2011)*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

18. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.
19. S. Kim, K. Yoon, J. Kwon, S. Hong, and Y.-H. Park. Efficient isogeny computations on twisted Edwards curves. *Security and Communication Networks*, 2018, 2018.
20. P. Koppermann, E. Pop, J. Heyszl, and G. Sigl. 18 seconds to key exchange: Limitations of supersingular isogeny diffie-hellman on embedded devices. Cryptology ePrint Archive, Report 2018/932, 2018. https://eprint.iacr.org/2018/932.
21. B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani. Fast hardware architectures for supersingular isogeny Diffie-Hellman key exchange on FPGA. In *International Conference in Cryptology in India*, pages 191–206. Springer, 2016.
22. B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani. NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In *International Conference on Cryptology and Network Security (CANS 2016)*, pages 88–103. Springer, 2016.
23. Z. Liu, P. Longa, G. Pereira, O. Reparaz, and H. Seo. Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks. In *International Conference on Cryptographic Hardware and Embedded Systems-CHES2017*, pages 665–686, 2017.
24. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
25. H. Seo, Z. Liu, P. Longa, and Z. Hu. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–20, 2018.
26. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.
27. The National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization, 2017–2018. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.