SIKE Round 2 Speed Record on Embedded Processors

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

IT Department, Hansung University, Seoul, South Korea, hwajeong84@gmail.com
Department of Computer and Electrical Engineering and Computer Science,
Florida Atlantic University, FL, USA,
{ajalali2016,razarderakhsh}@fau.edu

Abstract. We present the optimized software implementation of Supersingular Isogeny Key Encapsulation (SIKE) round 2, on low-end 32-bit ARM Cortex-M4 microcontrollers and high-end 64-bit ARM Cortex-A53 processors. The proposed library introduces a new speed record of SIKE protocol on the target embedded processors. 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. The benchmark result on STM32F4 Discovery board equipped with low-end 32-bit ARM Cortex-M4 microcontroller shows that the entire key encapsulation at NIST security level 1 (i.e. SIKEp434) takes about 252 million clock cycles (i.e. 1.5 seconds @168MHz). In contrast to the previous optimized implementation of the isogeny-based key exchange on low-power 32-bit ARM Cortex-M4, our result shows a feasibility of using SIKE mechanism on low-end microcontrollers. The SIKE round 2 key encapsulation mechanism on Odroid-C2 board equipped with high-end 64-bit ARM Cortex-A53@1.536GHz takes only 65 ms at NIST security level 1. Considering SIKE's extremely small key size in comparison to other NIST PQC round 2 candidates, our result implies that SIKE is one of the promising candidates for key encapsulation mechanism on low-end and high-end embedded devices in the quantum era.

Keywords: Post-quantum cryptography, SIKE, Montgomery multiplication, 32-bit ARM Cortex-M4, 64-bit ARM Cortex-A53

1 Introduction

Initiated by the National Institute of Standards and Technology (NIST), Post-Quantum Cryptography (PQC) has been elevated to a standardization process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms [30]. To prepare for security concerns caused by quantum computers, in 2016, NIST called for the cryptographic algorithms which were assumed to be resistance against high-scale quantum computers. These proposals provided key encapsulation mechanism (KEM) or digital signature algorithms from different arithmetic structures, resulting in different characteristics and parameters. Recently, NIST announced approved candidates for round

2 which are the most promising candidates, in terms of security, performance, and compatibility with current technology. For the key encapsulation mechanism, only 17 candidates made it through to the second round for being evaluated and analyzed from different perspectives.

Different PQC candidates are constructed on hard mathematical problems which are assumed to be impossible to solve even for large-scale quantum computers. These problems can be categorized into five main categories: code-based cryptography, lattice-based cryptography, hash-based cryptography, multivariate cryptography, and supersingular isogeny-based cryptography, see, for instance [10].

Supersingular Isogeny Key Encapsulation (SIKE) mechanism is one of the PQC candidates which is constructed on the hardness of solving isogeny maps between supersingular elliptic curves. In fact, SIKE is the only candidate that offers the quantum-resistance cryptographic construction over elliptic curves, resulting in well-known structures in implementation perspective. The proposed key encapsulation mechanism is derived from the original Jao-De Feo's Diffie-Hellman key-exchange and public-key encryption algorithms [20]. However, constructing cryptographic structures from hardness of supersingular isogeny graphs was introduced by Charels-Lauter-Goren [9].

The first round SIKE submission [6] offered three different security levels known as SIKEp503, SIKEp751, and SIKEp964. According to the best known quantum attacks on solving supersingular isogeny problem by that time, the proposed security levels met NIST's level 1, 3, and 5 requirements, respectively.

However, recent studies on the cost of solving isogeny problem on quantum computers by Adj et al. [1] revealed that the security assumptions for SIKE was too conservative. In fact, a set of realistic models of quantum computation on solving Computational Supersingular Isogeny (CSSI) problem in [1] suggests that the Oorschot-Wiener golden collision search is the most powerful attack on the CSSI problem, resulting in significant improvement on the SIKE's classical and quantum security levels.

Accordingly, the second round SIKE [4] offers a new set of security levels which are more realistic and provide significant improvement on the key encapsulation performance. In particular, decreasing the bit-length of SIKE's primes translates to notable performance improvement, making this scheme suitable for many potential applications on low-end and high-end embedded devices.

In CANS'16, Koziel et al. presented first SIDH implementations on 32-bit ARM Cortex-A processors [25]. In 2017, Jalali et al. presented first SIDH implementations on 64-bit ARM Cortex-A processors [18]. In CHES'18, Seo et al. improved previous SIDH and SIKE implementations on high-end 32/64-bit ARM Cortex-A processors [28]. At the same time, the implementations of SIDH on Intel and FPGA are also successfully evaluated [14, 8, 22, 24]. Afterward, in 2018, first implementation of SIDH on low-end 32-bit ARM Cortex-M4 microcontroller was suggested [23].

In this work, we provide a full report on the highly-optimized implementation of SIKE on low-end 32-bit and high-end 64-bit ARM embedded processors.

Our proposed library takes advantage of state-of-the-art engineering techniques as well as low level assembly optimizations. We studied different approaches for finite field arithmetic implementation over SIKE's new primes. Our benchmark results offer significant improvement in performance compared to previous implementations, suggesting the possible integration of this scheme on embedded processors in the future.

2 SIKE Round 2 on ARM Cortex-M4

2.1 ARM Cortex-M4 Architecture

With over 100 billion ARM-based chips shipped worldwide as of 2017 [2], ARM is the most popular instruction set architecture (ISA), in terms of quantity. In this work, we firstly target the popular low-end 32-bit ARM Cortex-M4 microcontrollers, which belong to the "microcontroller" profile implemented by cores from the Cortex-M series. The ARM Cortex-M architecture is a reduced instruction set computer (RISC) using a load-store architecture. The ARM Cortex-M4 microcontrollers support a three-stage pipeline, and memory accesses involving 1 register and n registers take 2 cycles and $n + 1$ cycles, respectively.

As other traditional 32-bit ARM architectures, the ARM Cortex-M4 ISA is equipped with 16 32-bit registers (`R0`∼`R15`), from which 15 (`R0`∼`R12, R13 (SP), R14 (LR)`) are available. `R13`, `R14`, and `R15` registers are reserved for stack pointer, link register, and program counter, respectively. The `R13` and `R14` registers can be freed up by saving it in slower memory and retrieving it after the register has been used.

Since the maximum capacity of the 15 registers is of only 480 bits (32×15), efficient use of the available registers to minimize the number of memory accesses is a critical strategy for optimized implementations of multi-precision multiplications (i.e. 512-bit and 768-bit). The ARM Cortex-M4 provides an instruction set supporting 32-bit operations or, in the case of Thumb and Thumb2, a mix of 16- and 32-bit operations. The instruction set is comprised of standard instructions for basic arithmetic (i.e. addition and addition with carry operations) and logic operations. However, in contrast to other lower processor classes, the ARM Cortex-M4 supports for the so-called DSP instructions, which include unsigned multiplication with double accumulation `UMAAL` instruction.

The `UMAAL` instruction performs a 32×32-bit multiplication followed by accumulations with two 32-bit values. This instruction achieves the same latency (i.e. 1 clock cycle) and throughput of the unsigned multiplication instruction, which means that accumulation (i.e. two 32-bit addition operations) is virtually executed for free. The detailed descriptions of multiplication operations are as follows:

- `UMULL` (unsigned multiplication):
  `UMULL R0, R1, R2, R3` computes (`R1` ‖ `R0`) ← `R2` × `R3`.
- `UMLAL` (unsigned multiplication with accumulation):
  `UMLAL R0, R1, R2, R3` computes (`R1` ‖ `R0`) ← (`R1` ‖ `R0`) + `R2` × `R3`.

– `UMAAL` (unsigned multiplication with double accumulation):
`UMAAL R0, R1, R2, R3` computes (`R1` $\|$ `R0`) $\leftarrow$ `R1 + R0 + R2` $\times$ `R3`.

The popularity of ARM Cortex-M4 microcontrollers in different applications introduced a post-quantum cryptography software library (`pqm4`) which targets this family of microcontrollers [21]. The `pqm4` library provides a framework for benchmarking and testing, started as a result of the PQCRYPTO project funded by the European Commission in the H2020 program. The library currently contains implementations of 10 post-quantum key-encapsulation mechanisms and 3 post-quantum signature schemes targeting the ARM Cortex-M4 family of microcontrollers. In particular, `pqm4` targets the STM32F4 Discovery board, featuring an ARM Cortex-M4 CPU@168MHz, 1MB of Flash, and 192KB of RAM. The library offers a simple build system that generates an individual static library for each implementation for each scheme. After compilation, the library provides automated benchmarking for speed and stack usage. As a result, we chose to evaluate the performance of our proposed library with `pqm4` framework to provide a fair and valid comparison with other PQC schemes.

In the following Section, we describe the proposed engineering techniques for designing highly-optimized arithmetic libraries, targeting different security levels of SIKE schemes on 32-bit ARM Cortex-M4 microcontrollers.

2.2 Multiprecision Multiplication

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

Figure 1 illustrate the 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, 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 ($i = 2n-1$).

Efficient register utilization The Operand Caching (OC) method follows the product-scanning approach for inner loop but it divides the calculation (i.e. outer loop) into several rows [17]. 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.

[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.

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 [11]	$2n^2$	$2n$
Hybrid Scanning [16]	$2\lceil n^2/d \rceil$	$2n$
Operand Caching [17]	$2\lceil n^2/e \rceil$	$\lceil n^2/e \rceil + n$
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

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 number of 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 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 intermediate 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

[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.

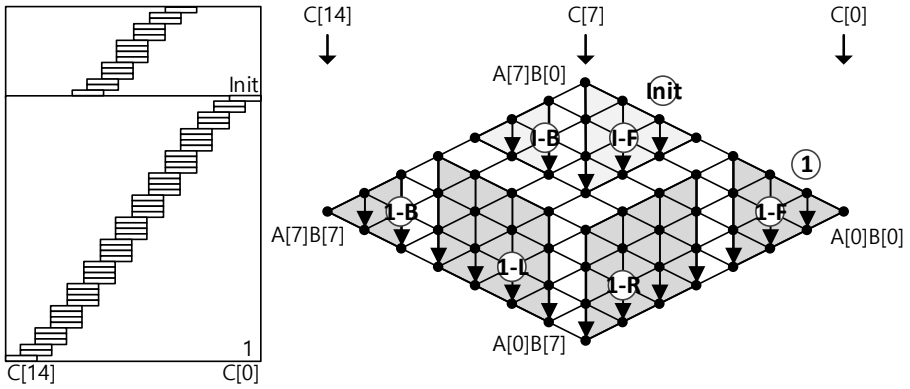

Fig. 1: Proposed 256-bit Refined Operand Caching multiplication at the word-level where e is 4 on ARM Cortex-M4, Ⓘnit: initial block; ①: order of rows; Ⓕ: front part; Ⓡ: middle right part; Ⓛ: middle left part; Ⓑ: back part.

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.

Optimized front parts As it is illustrated in Figure 1, 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 row`1`, 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.

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 [15]. Moreover, the intermediate results are efficiently handled with carry-less MAC routines by using the `UMAAL` instructions. Figure 2 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

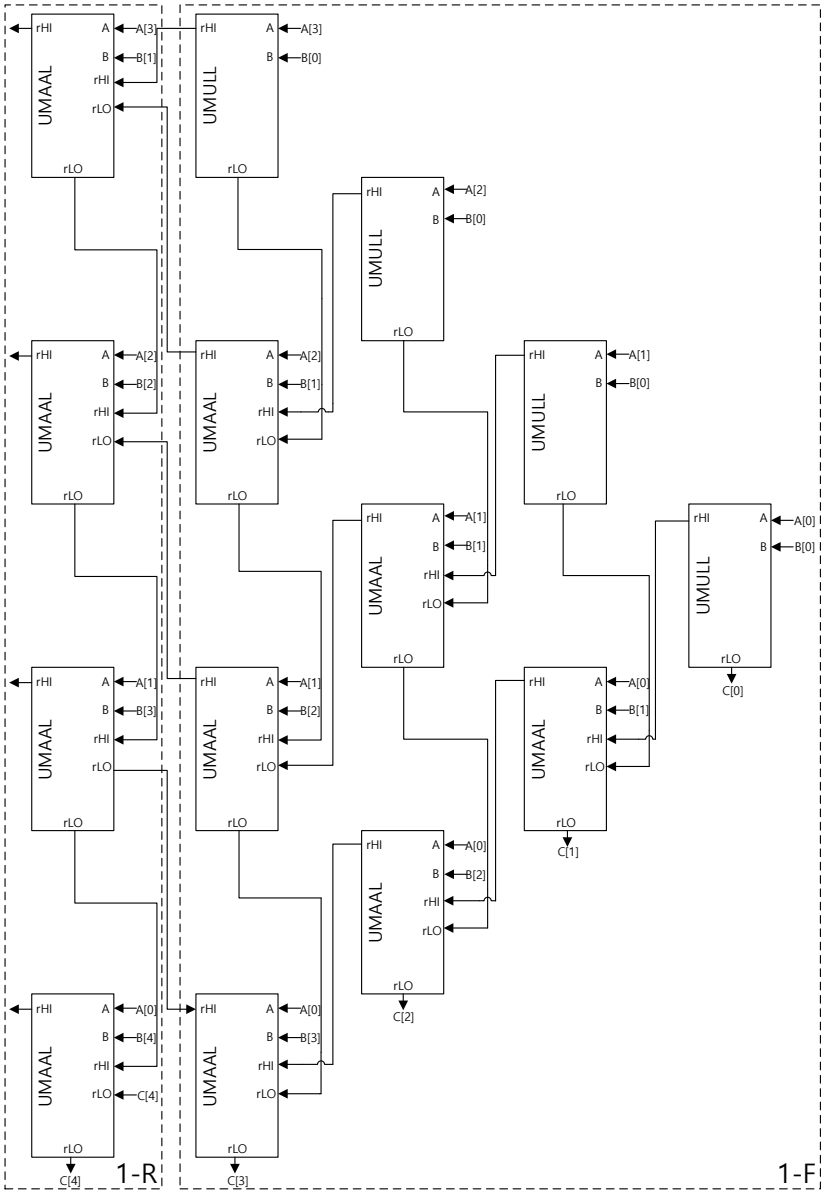

Fig. 2: 4-word integers with the product scanning approach using the `UMULL` and `UMAAL` instructions for front part of OC method.

in Figure 2 (`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.

2.3 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 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

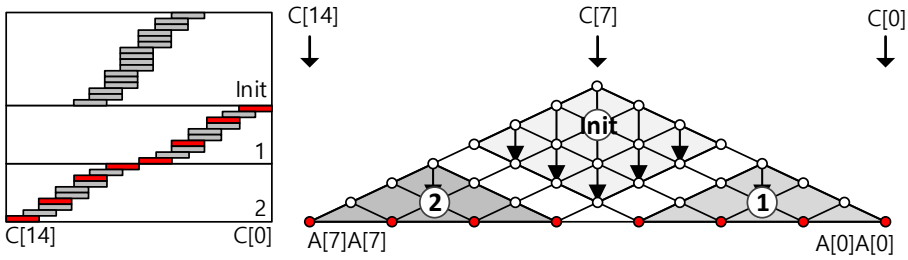

Fig. 3: 255-bit proposed squaring at the word-level on ARM Cortex-M4, Ⓘnit: initial block; ① → ②: order of rows.

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 3 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.

2.4 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 [26]. We adapt the implementation techniques described in sections 2.2 and 2.3 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 [12, 5]. 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. [12] 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 [13].

In CHES'18, Seo et al. suggested the variant of Hybrid-Scanning (HS) for "SIDH-friendly" Montgomery reduction on ARM Cortex-A15 [28]. Similar to OC method, the HS method also changes the operand pointer when the row is changed. By using the register utilization described in Section 2.2, we increase the parameter d by 1 ($3 \rightarrow 4$. Moreover, the initial block is also optimized

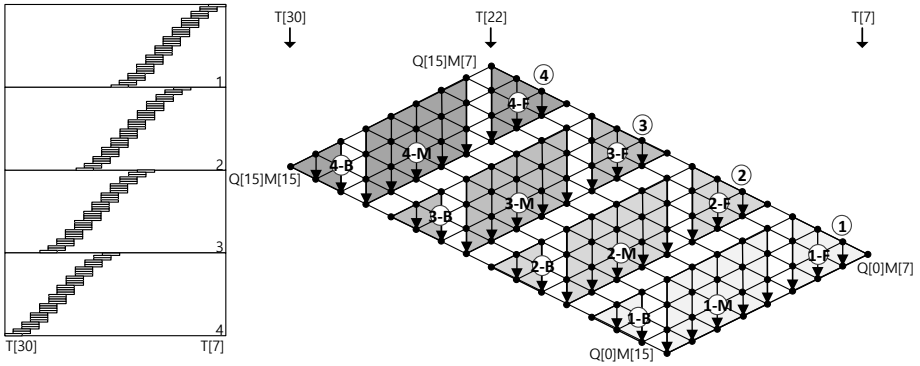

Fig. 4: 503-bit "SIDH-friendly" Montgomery reduction at the word-level, where d is 4 on ARM Cortex-M4, ① → ② → ③ → ④: order of rows; Ⓕ: front part; Ⓜ: middle part; Ⓑ: back part; where M, R, T, and Q are modulus, Montgomery radix, intermediate results, and quotient ($Q \leftarrow T \cdot M' \bmod R$).

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 4, 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.

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

2.5 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 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 [5] 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`.[3]

Table 3 and 4 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. [13] and the reference C implementation of SIKE [5] 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 [23] by 2.75x and 4.35x, respectively. The implementations of SIKEp434, SIKEp503, and SIKEp751 also show better performance than previous works[4]. In particularly, the entire key encapsulation at NIST security level 1 takes about 252 million clock cycles (i.e. 1.5 seconds @168MHz)

Compared with other NIST PQC round 2 schemes, the SIKE protocol shows relatively slower execution time but the SIKE protocols show the most competitive memory utilization for encapsulation and decapsulation. Furthermore, small key size of SIKE ensures the lower energy consumption for key transmission (through wireless network) than other schemes. The low-energy consumption is the most critical requirement for low-end (battery-powered) microcontrollers.

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

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

Table 3: 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 [13]	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 [13]	C	1,579	1,092	55,178	–	3,246	3,651	2,669	3,112	12,678
Koppermann et al. [23]	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 4: Comparison of SIKEp434, SIKEp503, and SIKEp751 protocols on the ARM Cortex-M4 microcontrollers. Timings are reported in terms of clock cycles. Koppermann et al. [23] 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 [13]	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 [13]	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

3 SIKE Round 2 on ARMv8 Cortex-A

3.1 ARMv8 Cortex-A Architecture

ARMv8 Cortex-A, or simply ARMv8, is the latest generation of ARM architectures targeted at the "application" profile. It includes the typical 32-bit architecture, called "AArch32", and advanced 64-bit architecture named "AArch64" with its associated instruction set "A64" [3]. AArch32 preserves backwards compatibility with ARMv7 and supports the so-called "A32" and "T2" instructions sets, which correspond to the traditional 32-bit and Thumb instruction sets, respectively. AArch64 comes equipped with 31 general purpose 64-bit registers (i.e. `X0`∼`X30`) and one zero register (i.e. `XZR`), and an instruction set supporting 32-bit and 64-bit operations. The significant register expansion means that with AArch64 the maximum register capacity is expanded to 1,984 bits (i.e. 31×64, a 4x increase with respect to ARMv7.).

ARMv8 processors started to dominate the smartphone market soon after their first release in 2011, and nowadays they are widely used in various high-end smartphones (e.g. iPhone, Huawei Mate, and Samsung Galaxy series). Since this architecture is used primarily in embedded systems and smartphones, efficient and compact implementations are of special interest.

ARMv8 processor supports powerful 64-bit wise unsigned integer multiplication instructions. Our implementation of modular multiplication uses the AArch64 architecture and makes extensive use of the following multiply instructions:

- `MUL` (unsigned multiplication, low part):
  `MUL X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`) mod 2^{64}.
- `UMULH` (unsigned multiplication, high part):
  `UMULH X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`)$/2^{64}$.

The two instructions above are required to compute a full 64-bit multiplication of the form 128-bit $\leftarrow 64 \times 64$-bit, namely, the `MUL` instruction computes the lower 64-bit half of the product while `UMULH` computes the higher 64-bit half.

For the addition and subtraction operations, `ADDS` and `SUBS` instructions ensure 64-bit wise results, respectively. The detailed descriptions are as follows:

- `ADDS` (unsigned addition):
  `ADDS X0, X1, X2` computes {`CARRY,X0`} $\leftarrow$ (`X1` + `X2`).
- `SUB` (unsigned subtraction):
  `SUBS X0, X1, X2` computes {`BORROW,X0`} $\leftarrow$ (`X1` − `X2`).

3.2 Multiprecision Multiplication

There is a number of works in the literature that study the ARMv8 instructions to implement multi-precision multiplication or the full Montgomery multiplication for "SIDH friendly" modulus [19, 18, 29]. In [18], Jalali et al. implemented 751-bit and 964-bit finite field multiplication. They utilized the Comba method (i.e. column-wise multiplication) for both cases [11]. In particular, they used 2-level Karatsuba for 964-bit finite field multiplication, which shows 23.9% performance enhancements than conventional Comba method. In [29], Seo et al. optimized the 503-bit finite field multiplication for SIKEp503. They also used the Comba method with 2-level Karatsuba method to enhance the performance of multiplication. Furthermore, they optimized the MAC (Multiplication ACcumulation) routines to avoid the pipeline stalls.

Recently, two novel SIKE protocols (i.e. SIKEp434 and SIKEp610) for NIST Post Quantum Cryptography competition were suggested, which meet NIST security level 1 and 3, respectively [4]. However, previous works do not show the optimized results for both protocols. In this paper, we show the first practical implementations of SIKEp434 and SIKEp610 protocols on 64-bit ARMv8-A processors.

In previous works, they used the Comba method (i.e. column-wise method) to improve the multi-precision multiplication. The Comba method performs the partial products in column-wise, which ensures small number of registers for maintaining the intermediate results. In Figure 5, the part of Multiplication ACculmuation (MAC) routine in column-wise method for 64-bit ARMv8 processors is described. The example performs the three partial products ($A[i] \times B[j]$, $A[i+1] \times B[j-1]$, and $A[i+2] \times B[j-2]$) and accumulates them to the intermediate results. In each MAC routine, two multiplication (`MUL_LOW` and `MUL_HIGH`)

and three addition operations (`ACC0`, `ACC1`, and `ACC2`) are required. For one limb multiplication, we need three addition operations. For that reason, n-limb multiplication requires $3 \times n^2$ addition operations.

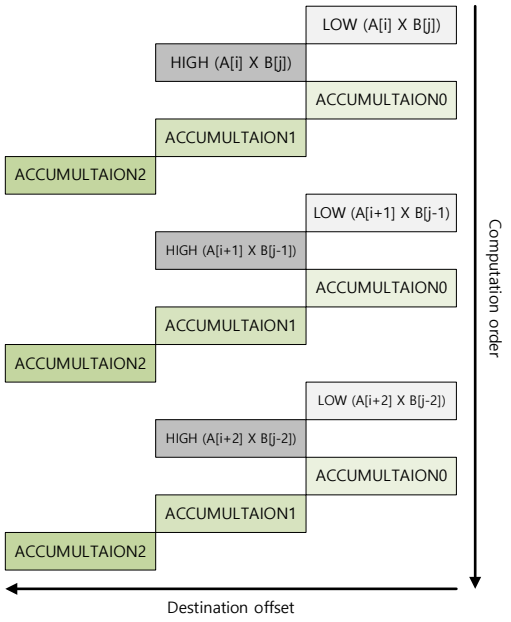

Fig. 5: Part of column-wise multiplication for ARMv8

In this work, we target the relatively shorter modulus (i.e. 434-bit) than previous works (i.e. 503-bit or 751-bit). We decide to use the row-wise multiplication, which requires $2n + 2$ registers ($n + 1$ for operands and $n + 1$ for intermediate results), where n, m, and w are $\lfloor m/w \rfloor$, operand length, and word size, respectively. Under 64-bit processor setting, the n is set to 7 for 434-bit ($\lfloor 434/64 \rfloor$). Considering that ARMv8 supports 31 64-bit registers, the required number of registers for 434-bit can be retained in the registers. In Figure 6, the part of MAC routine in row-wise method for 64-bit ARMv8 processors is described. The example performs the three partial products ($A[i] \times B[j]$, $A[i] \times B[j + 1]$, and $A[i] \times B[j + 2]$) and accumulates them to the intermediate results. The number of addition for three partial products in Figure 6 are 8 (i.e. $2 \times (n+1)$ where n is 3.). For the n-limb multiplication, it requires $2 \times n \times (n+1)$ addition operations. The comparison of multiplication methods in terms of the number of addition operations depending on the number of limb are given in Table 5. Compared with the column-wise method (i.e. product-scanning), the row-wise method (i.e. operand-scanning) requires less number of addition operations for accumulation routines. For the 7-limb case (i.e. 434-bit), the row-wise method reduces

Table 5: Comparison of multiplication methods, in terms of the number of addition operations depending on the number of limb.

Method	3	4	5	6	7
Operand Scanning	24	40	60	84	112
Product Scanning	27	48	75	108	147

the number of addition operations by 35 times than the column-wise method. The multiplication is performed in original row-wise multiplication rather than row-wise multiplication with Karatsuba method. The Karatsuba method is also working for 7-limb case but it generates a number of sub-routines to perform and store the intermediate results, which requires additional operations and memory accesses [27].

For the 610-bit multiplication, the operands $A = (A[9], \ldots, A[0])$ and $B = (B[9], \ldots, B[0])$ need 20 64-bit registers. Except the operands, we also need registers for intermediate results and temporal storage. Due to the limited number of registers, we only maintain the half number of operands in the registers and load the remaining operands on demand.

We first compute the lower 320-bit multiplication $R_L \leftarrow A[4 \sim 0] \cdot B[4 \sim 0])$ using the row-wise method that requires 25 `MUL`, 25 `UMULH` and 52 addition instructions for accumulating the partial products. Second, we compute the higher 310-bit multiplication $R_H \leftarrow A[9 \sim 5] \cdot B[9 \sim 5]$, similarly. Third, we compute the subtractions and absolute values $|A[4 \sim 0] - A[9 \sim 5]|$ and $|B[4 \sim 0] - B[9 \sim 5]|$ and proceed to the last 310-bit multiplication $R_M \leftarrow |A[4 \sim 0] - A[9 \sim 5]| \cdot |B[4 \sim 0] - B[9 \sim 5]|$. Finally, we obtain the result by performing the accumulation step $R_H \cdot 2^{610} + (R_L + R_H - R_M) \cdot 2^{310} + R_L$. Since the multiplication uses all available registers, 12 callee-saved registers ($X19 \sim X30$) are stored into the stack. The multiplication is also designed to reduce the pipeline stalls. The multiplication and addition/subtraction operations use different instruction group. They can hide each others costs. Based on the above observation, we engineer a multi-precision multiplication to hide the addition costs into the multiplication. At the lowest level, we implement multi-precision multiplication using the row-wise method based on the following multiplication/addition instruction sequence:

```
  ⋮
MUL  X7, X6, X2
ADCS X18, X18, X13
MUL  X8, X6, X3
ADCS X19, X19, X14
MUL  X9, X6, X4
ADCS X20, X20, X15
MUL  X10, X6, X5
ADCS X21, X21, X16
  ⋮
```

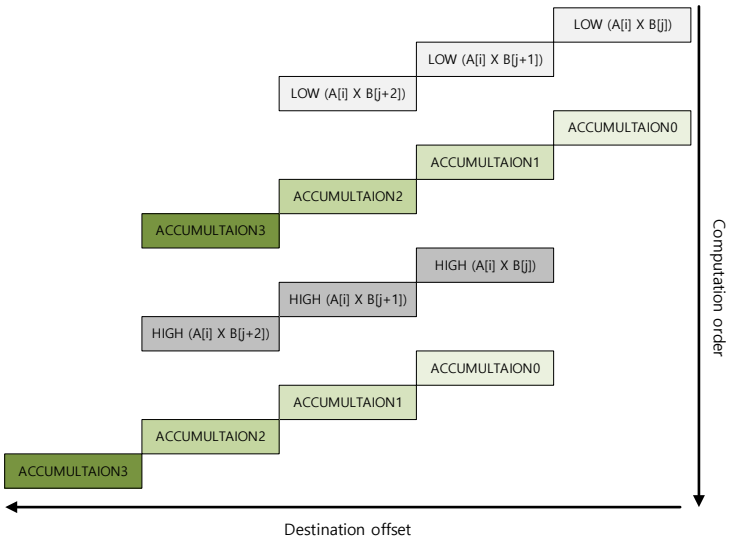

Fig. 6: Part of row-wise multiplication for ARMv8

We ensure that the destination of `MUL` instruction is not used for the source of following `ADCS` instructions. This approach avoids the pipeline stalls. Second, `MUL` and `ADCS` instructions are performed one by one to hide the each costs.

3.3 Modular Reduction

In this section, we adapt the techniques described in previous sections to implement modular multiplication for the supersingular isogeny-based protocols SIDH and SIKE. Specifically, we target the parameter sets based on the primes p434 and p610 [4].

Multi-precision modular multiplication is the most expensive operation for the implementation of SIKE [20, 12]. In particular, Montgomery multiplication for SIKE can be efficiently exploited and further simplified by taking advantage of so-called "Montgomery-friendly" modulus. The advantage of using Montgomery multiplication for "SIDH-friendly" primes was recently confirmed by Bos and Friedberger [7], who studied and compared different approaches, including Barrett reduction. Recent works by Seo et al also utilized the Montgomery multiplication for SIKEp503 protocols [29].

Based on the observation above, we choose the Montgomery multiplication to implement SIDH-friendly modular arithmetic for SIKEp434 and SIKEp610 protocols. The approach reduces almost half of partial products since the lower part is set to 0. In order to reduce the memory accesses, we keep as many results as possible in the registers. Since the Montgomery multiplication performs the partial products with modulus and quotient (Quotient is intermediate results

multiplied by constant m'), we maintained all quotients in the registers and used them directly. The technique reduces the $2 \times (n + 1)$ number of memory accesses for $n + 1$ load and $n + 1$ store operations.

3.4 Performance Evaluation

In this section, we evaluate the performance of the proposed algorithms for 64-bit ARMv8-A processors. All our implementations were written in assembly language and complied with optimization level `-O3`.

We implemented the multi-precision multiplication algorithm described in Section 3.2 and Montgomery reduction in Section 3.3. We integrated our implementation of the Montgomery multiplication for ARMv8-A into the SIKE round 2 library [4].

Table 6 summarizes the results of different software implementations of the SIKEp434 and SIKEp610 arithmetic on ARMv8-A processor: a 1.536GHz ARM Cortex-A53 processor. Since this is first work for SIKEp434 and SIKEp610 on ARMv8-A processors, we compare the results with the SIKE round 2 reference code. The *unoptimized* reference implementation is written in C using the SIKE round 2 library [4]. In this case, the proposed arithmetic implementations show much higher performance than reference work. In particular, finite field multiplication and inversion operations show performance enhancements by 4.96x and 4.98x, respectively.

Table 7 summarizes the results of different software implementations of the SIKEp434 and SIKEp610 protocols on ARMv8-A processor. Compared with reference work, the proposed implementation is between 3.83 and 3.42 times faster for the computation of the SIKE full protocols. Considering that the target processor is 1.536 GHZ, the SIKEp434 and SIKEp610 requires only 0.065 and 0.238 seconds, respectively.

Compared with the other security levels, the performance depends on the length of modulus. The SIKEp434 shows the highest performance and the SIKEp751 shows the lowest performance as we expected.

Table 6: Comparison of implementations of the SIKEp434 and SIKEp610 arithmetic on ARMv8 Cortex-A53 based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Protocol	Timings [cc]			
			$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ inv
SIKE R2 [4]	C	SIKEp434	172	129	3,110	1,648,372
This work	ASM		71	63	691	380,711
SIKE R2 [4]	C	SIKEp610	257	187	6,599	4,800,694
This work	ASM		100	91	1,329	963,064

Table 7: Comparison of implementations of the SIKE protocols on ARMv8 Cortex-A53 based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Protocol	Timings [cc]	Timings [$cc \times 10^6$]			
			$\mathbb{F}_p$ mul	KeyGen	Encaps	Decaps	Total
SIKE R2 [4]	C	SIKEp434	3,110	114	186	199	499
This work	ASM		691	30	49	52	130
Seo et al. [29]	ASM	SIKEp503	849	38	63	67	168
SIKE R2 [4]	C	SIKEp610	6,599	344	634	615	1,593
This work	ASM		1,329	99	183	183	465
Seo et al. [29]	ASM	SIKEp751	2,450	164	265	284	713

4 Conclusion

In this work, we presented a highly optimized implementation of SIDH/SIKE on low-end 32-bit ARM Cortex-M4 microcontrollers and high-end 64-bit ARM Cortex-A53 embedded processors. We proposed a new set of implementation techniques, taking advantage of Cortex-M4 and Cortex-A53 capabilities. In particular, we proposed a new implementation method for finite field arithmetic implementation.

We integrated the proposed modular arithmetic implementations into SIDH/SIKE reference implementations. Our library significantly outperforms the previous state-of-the-art implementations of integer arithmetic on our target platform, providing 4.35x faster results compared to the only available optimized implementation of SIDHp751 on Cortex-M4 in the literature. Using our proposed techniques and optimizations, the entire key encapsulation mechanism over SIKEp434 runs in 1.5 seconds on a 168MHz ARM Cortex-M4 microcontroller which shows the feasibility of using post-quantum isogeny-based cryptography on low-end microcontrollers.

The optimized implementation on a 64-bit ARMv8 Cortex-A53 processors, which push further the performance of post-quantum supersingular isogeny-based protocols, are 3.42x faster than the previously implementations of SIDHp610 on the same processors. Furthermore, we integrated our fast modular arithmetic implementations, compact prime SIDHp434, and optimal strategy for isogeny computations into Microsoft's SIDH library. A 128-bit full key-exchange execution over optimal prime SIDHp434 is performed in about 0.065 seconds on a 1.536GHz ARMv8 Cortex-A53 processors.

We hope the proposed implementation techniques motivate more engineering efforts on the optimized implementation of SIKE mechanism on different embedded platforms. We plan to adopt the same strategy in designing efficient software libraries, targeting different families of embedded processors in the future.

5 Acknowledgement

This work was supported by Institute for Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (<Q|Crypton>, No.2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity).

References

1. G. Adj, D. Cervantes-Vázquez, J.-J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. Technical report, Cryptology ePrint Archive, Report 2018/313, 2018. https://eprint.iacr.org, 2018.
2. ARM Holdings. Q1 2017 roadshow slides. `https://www.arm.com/company/-/media/arm-com/company/Investors/Quarterly%20Results%20-%20PDFs/Arm_SB_Q1_2017_Roadshow_Slides_Final.pdf`, 2017.
3. ARM Limited. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`, 2013–2017.
4. 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, 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`.
5. 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, 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`.
6. 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`.
7. J. Bos and S. Friedberger. Fast arithmetic modulo $2^x p^y \pm 1$. In *IEEE Symposium on Computer Arithmetic (ARITH'17)*, pages 148–155. IEEE, 2017.
8. J. Bos and S. Friedberger. Arithmetic considerations for isogeny based cryptography. *IEEE Transactions on Computers*, 2018.
9. D. X. Charles, K. E. Lauter, and E. Z. Goren. Cryptographic hash functions from expander graphs. *J. Cryptology*, 22(1):93–113, 2009.
10. L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.
11. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM systems journal*, 29(4):526–538, 1990.

12. 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.
13. C. Costello, P. Longa, and M. Naehrig. SIDH Library. `https://github.com/Microsoft/PQCrypto-SIDH`, 2016–2018.
14. 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.
15. H. Fujii and D. F. Aranha. Curve25519 for the Cortex-M4 and beyond. *Progress in Cryptology-LATINCRYPT*, 35:36–37, 2017.
16. 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.
17. 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.
18. 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.
19. A. Jalali, R. Azarderakhsh, and M. Mozaffari-Kermani. Efficient post-quantum undeniable signature on 64-bit ARM. In *International Conference on Selected Areas in Cryptography*, pages 281–298. Springer, 2017.
20. 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.
21. 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`.
22. 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.
23. 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`.
24. 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.
25. 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.
26. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.
27. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
28. 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.

29. 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.
30. 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`.