Optimized Implementation of SIKE Round 2 on 64-bit ARM Cortex-A Processors

Hwajeong Seo, Pakize Sanal, Amir Jalali, and Reza Azarderakhsh, *Member, IEEE*

***Abstract*— In this work, we present the first highly-optimized implementation of Supersingular Isogeny Key Encapsulation (SIKE) submitted to NIST's second round of post quantum standardization process, on 64-bit ARMv8 processors. To the best of our knowledge, this work is the first optimized implementation of SIKE round 2 on 64-bit ARM over SIKEp434 and SIKEp610. The proposed library is explicitly optimized for these two security levels and provides constant-time implementation of the SIKE mechanism on ARMv8-powered embedded devices. We adapt different optimization techniques to reduce the total number of underlying arithmetic operations on the field level. In particular, benchmark results on embedded processors equipped with ARM Cortex-A55@1.766GHz and ARM Cortex-A75@2.803GHz show that the entire SIKE round 2 Key Encapsulation Mechanism (KEM) takes only 98.6 ms and 85.3ms at NIST's security level 1, respectively. We also evaluated the compressed version of NIST's security level 1, which requires 134.7 ms and 113.7 ms for Cortex-A55 and Cortex-A75, respectively. Considering SIKE's extremely small key size in comparison to other post-quantum cryptography candidates, our result implies that SIKE is one of the promising candidates for key encapsulation mechanism on embedded devices in the quantum era.**

***Index Terms*— Post-quantum cryptography, isogeny-based cryptography, ARM processors, assembly, key encapsulation mechanism.**

I. 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 [25]. 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 cryptography 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 [9].

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 [8].

The first round SIKE submission 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. [2] 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 [2] 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.

Manuscript received November 24, 2019; revised February 14, 2020; accepted March 2, 2020. Date of publication March 16, 2020; date of current version July 31, 2020. This work of Hwajeong Seo was supported in part by the Institute for Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korean Government (MSIT) (Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity) under Grant 2019-0-00033, and in part by the Institute for Information & Communications Technology Promotion (IITP) Grant funded by the Korean Government (MSIT) (Research on Blockchain Security Technology for IoT Services) under Grant 2018-0-00264. This work of Reza Azarderakhsh was supported in part by NSF under Grant CNS-1801341, Grant NIST-60NANB17D184, Grant NIST-60NANB16D246, and Grant ARO W911NF-17-1-0311. This article was recommended by Associate Editor M. Martina. *(Corresponding author: Reza Azarderakhsh.)*

Hwajeong Seo is with the College of IT Engineering, Hansung University, Seoul 13557, South Korea (e-mail: hwajeong84@gmail.com).

Pakize Sanal and Reza Azarderakhsh are with the Department of Computer, Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431 USA (e-mail: psanal2018@fau.edu; razarderakhsh@fau.edu).

Amir Jalali is with the Information Security Group, LinkedIn Corporation, Sunnyvale, CA 94085 USA (e-mail: ajalali@linkedin.com).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2020.2979410

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 embedded devices.

In this work, we provide a full report on the highly-optimized implementation of SIKE on 64-bit ARM embedded processors over all proposed security levels. In particular, the reference optimized implementation of SIKE [4] on 64-bit ARM embedded processor only targets two security levels, i.e., SIKEp503 and SIKEp751. Therefore, in this work, we address this shortcoming by providing the KEM full benchmarks on different security levels which provide a reference for the performance analysis of this scheme for the second round.

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 portable implementation, suggesting the possible integration of this scheme on mobile devices in the future.

II. Background

In this section, we briefly review the SIDH protocol and the required steps for Alice and Bob to generate a shared secret. Furthermore, we describe the SIKE, a post-quantum key encapsulation mechanism from isogenies of supersingular elliptic curves which was submitted to NIST's PQC standardization competition. We refer the readers to [5], [20] for further details.

Let E_1 and E_2 be elliptic curves over a finite field $\mathbb{F}_q$. An isogeny $\phi : E_1 \rightarrow E_2$ is a non-constant rational map defined over $\mathbb{F}_q$ which is also a group homomorphism form $E_1(\mathbb{F}_q)$ to $E_2(\mathbb{F}_q)$. If such a map exists we say E_1 is isogenous to E_2, and two curves E_1 and E_2 over $\mathbb{F}_q$ are isogenous if and only if $\#E_1(\mathbb{F}_q) = \#E_2(\mathbb{F}_q)$.

An isogeny ϕ can be expressed in terms of two rational maps f and g over $\mathbb{F}_q$ such that $\phi((x, y)) = (f(x), y \cdot g(x))$. We can write $f(x) = p(x)/q(x)$ with polynomials $p(x)$ and $q(x)$ over $\mathbb{F}_q$ that do not have a common factor, and similarly for $g(x)$. The degree $deg(\phi)$ of the isogeny is defined as $max\{deg(p(x)), deg(q(x))\}$.

Given an isogeny $\phi : E_1 \rightarrow E_2$ we define the kernel of ϕ as follows:

$$ker(\phi) = \{P \in E_1 : \phi(P) = \mathcal{O}\}.$$

For any finite subgroup H of $E(\mathbb{F}_q)$, there is a unique isogeny $\phi : E \rightarrow E'$ such that $ker(\phi) = H$ and $deg(\phi) = |H|$, where $|H|$ denotes the cardinality of H. In this case, we denote by E/H the curve E'. Given a subgroup $H \subseteq E(\mathbb{F}_q)$, Velu formula can be used to find the isogeny ϕ and isogenous curve E/H. An example of isogeny map is given in Figure 1.

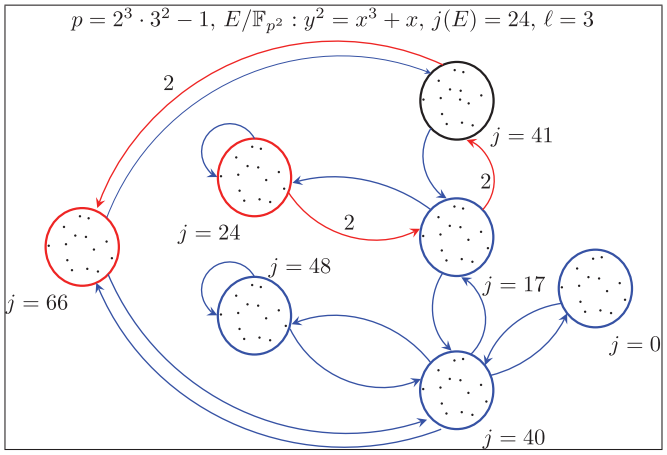

Fig. 1. An 9-degree isogeny map from elliptic curve with $j = 24$ to elliptic curve with $j = 17$ (each circle represents the isomorphism class, the numbers inside of the each circle are the j-invariants of that isomorphism class, the numbers next to the arrows denote the degree of the isogeny).

A. SIDH Key Exchange

In 2011, Jao and De Feo [20] proposed the SIDH, a quantum resistant key exchange protocol from isogenies of supersingular elliptic curves. Similar to classical Diffie-Hellman key exchange, SIDH protocol is constructed over some public parameters which are agreed upon by communication parties prior to key exchange.

1) Public Parameters: Fix a prime p of the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, e_A and e_B are positive integers, and f is a very small cofactor. We define a based supersingular elliptic curve E over $\mathbb{F}_{p^2}$ with cardinality $\#E = (\ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \mp 1)^2$, and base points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ from the torsion subgroups $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively, such that $\langle P_A, Q_A\rangle = E[\ell_A^{e_A}]$ and $\langle P_B, Q_B\rangle = E[\ell_B^{e_B}]$.

2) Key Exchange Protocol: Alice randomly chooses two integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, not both divisible by ℓ_A as her secret key and computes an isogeny $\phi_A : E \rightarrow E_A$ using kernel $R_A := \langle [m_A]P_A + [n_A]Q_A\rangle$. Alice also computes the image points $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$ by applying her secret isogeny ϕ_A to the public basis P_B and Q_B. She sends $\phi_A(P_B), \phi_A(Q_B)$ and E_A to Bob as her public key. Bob also selects random elements $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$, not both divisible by ℓ_B and computes a secret isogeny $\phi_B : E \rightarrow E_B$ from kernel $R_B := \langle [m_B]P_B + [n_B]Q_B\rangle$, along with image points $\{\phi_B(P_A), \phi_B(Q_A)\} \subset E_B$. He sends his public key, i.e., $\phi_B(P_A), \phi_B(Q_A)$ and E_B to Alice.

In the second round of key exchange, Alice uses Bob's public key $(\phi_B(P_A), \phi_B(Q_A), E_B)$ and computes an isogeny $\phi'_A : E_B \rightarrow E_{AB}$ from kernel equal to $\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A)\rangle$; Similarly, Bob computes an isogeny $\phi'_B : E_A \rightarrow E_{BA}$ having kernel $\langle [m_B]\phi_A(P_B)+[n_B]\phi_A(Q_B)\rangle$ using Alice's public key. Since the common j-invariant of E_{AB} and E_{BA} are equal, they use this value to form a secret shared key. The entire SIDH key exchange protocol is illustrated in Figure 2.

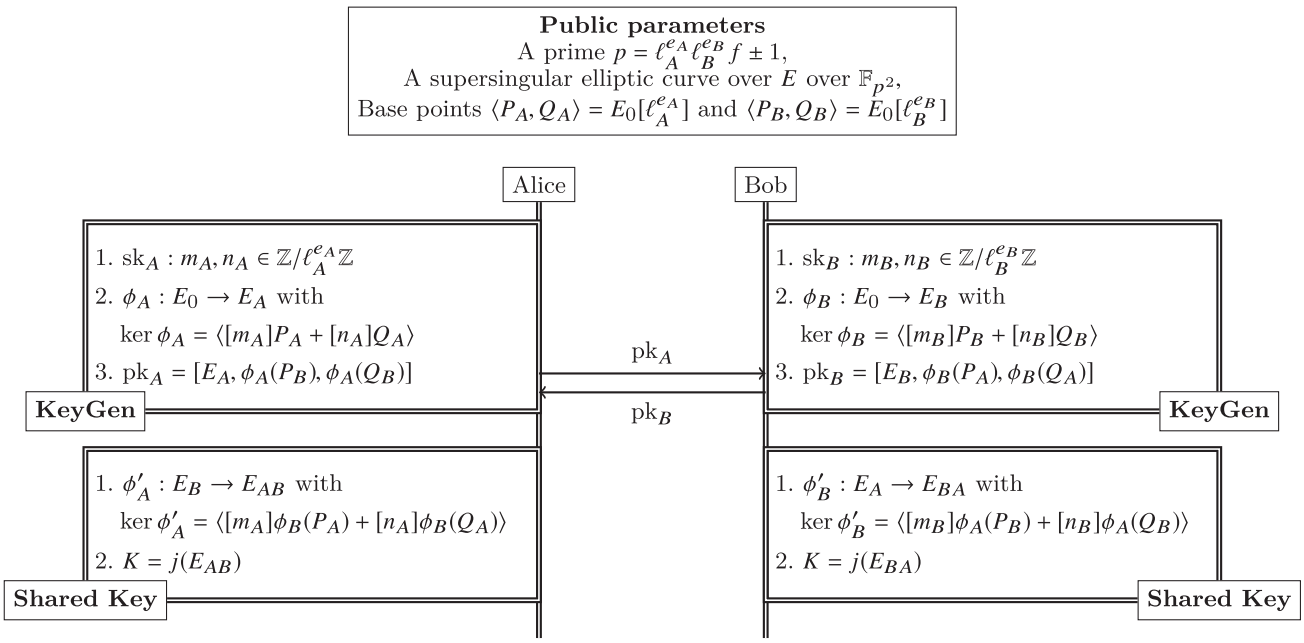

Fig. 2. SIDH key exchange protocol.

B. SIKE Mechanism

SIKE mechanism is constructed by applying a transformation of Hofheinz, Hövelmanns, and Kiltz [16] to the supersingular isogeny Public Key Encryption (PKE) scheme described in [20]. It is an actively secure key encapsulation mechanism (IND-CCA KEM) which addresses the static key vulnerability of SIDH due to active attacks in [14].

1) Public Parameters: Similar to SIDH, SIKE can be defined over a prime of the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$. However, for efficiency reasons, $\ell_A = 2, \ell_B = 3$, and $f = 1$ are fixed, thus the SIKE prime has the form of $p = 2^{e_A} \cdot 3^{e_B} - 1$. The starting supersingular elliptic curve $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$ with cardinality equal to $(2^{e_A} \cdot 3^{e_B})^2$, along with base points $\langle P_A, Q_A\rangle = E_0[2^{e_A}]$ and $\langle P_B, Q_B\rangle = E_0[3^{e_B}]$ are defined as public parameters.

2) Key Encapsulation Mechanism: The key encapsulation mechanism can be divided into three main operations: Alice's key generation, Bob's key encapsulation, and Alice's key decapsulation. We describe each operation in the following. Figure 3 presents the entire key encapsulation mechanism in a nutshell.

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

b) Encapsulation: Bob generates an t-bit random message $\mathrm{m} \in_R \{0,1\}^t$, concatenates it with Alice's public key pk_A and computes an $(e_B \log_2 3)$-bit hash value r using `cSHAKE256` hash function H_1, taking $m \parallel \mathrm{pk}_A$ as the input. Using r, he applies a secret isogeny $\phi_B : E_0 \to E_B$ to the base points $\{P_A, Q_A\}$ and forms his public key $\mathrm{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$. Bob also computes the common j-invariant of curve E_{BA} by applying another isogeny $\phi'_B : E_A \to E_{BA}$ using Alice's public key. Bob forms a ciphertext $c = (c_0, c_1)$, such that:

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

where H_2 is a `cSHAKE256` hash with a custom length output and a defined initialization parameter. Finally, Bob computes the shared secret as $K = H_3(m \parallel c)$ and sends c to Alice.

c) Decapsulation: Upon receipt of c, Alice computes the common j-invariant of E_{AB} by applying her secret isogeny to E_B. She computes $m' = c_1 \oplus H_2(j(E_{AB}))$ and $r' = H_1(m \parallel \mathrm{pk}_A)$. Finally, she validates Bob's public key by computing $\mathrm{pk}_B(r')$ and comparing it with c_0. She generates the same shared secret $K = H_3(m' \parallel c)$ if the public key is valid, otherwise she outputs a random value $K = H_3(s \parallel c)$ to be resistant against active attacks.

C. Key Compression

Compared to other PQC candidates, SIKE provides the smallest public key size which makes this scheme a suitable candidate for the applications with limited bandwidth/memory. Furthermore, the standard representation of SIKE's public keys can be further compressed using the key-compression technique proposed in the SIKE round 2 proposal with a slight but not negligible overhead in overall performance. First key compression is proposed by Azarderakhsh et al. in [6] and then it is improved in [11] for SIDH. Recently a compressed key for SIKE is appeared in SIKE round 2 in [4]. The key compression efficiency on SIKE has been improved by dual isogenies method. In particular, recent optimization in pairing computations and basis generation, in addition to fast and compact x-only formulas for dual isogenies of

[1]The value of t is defined by the implementation parameters.

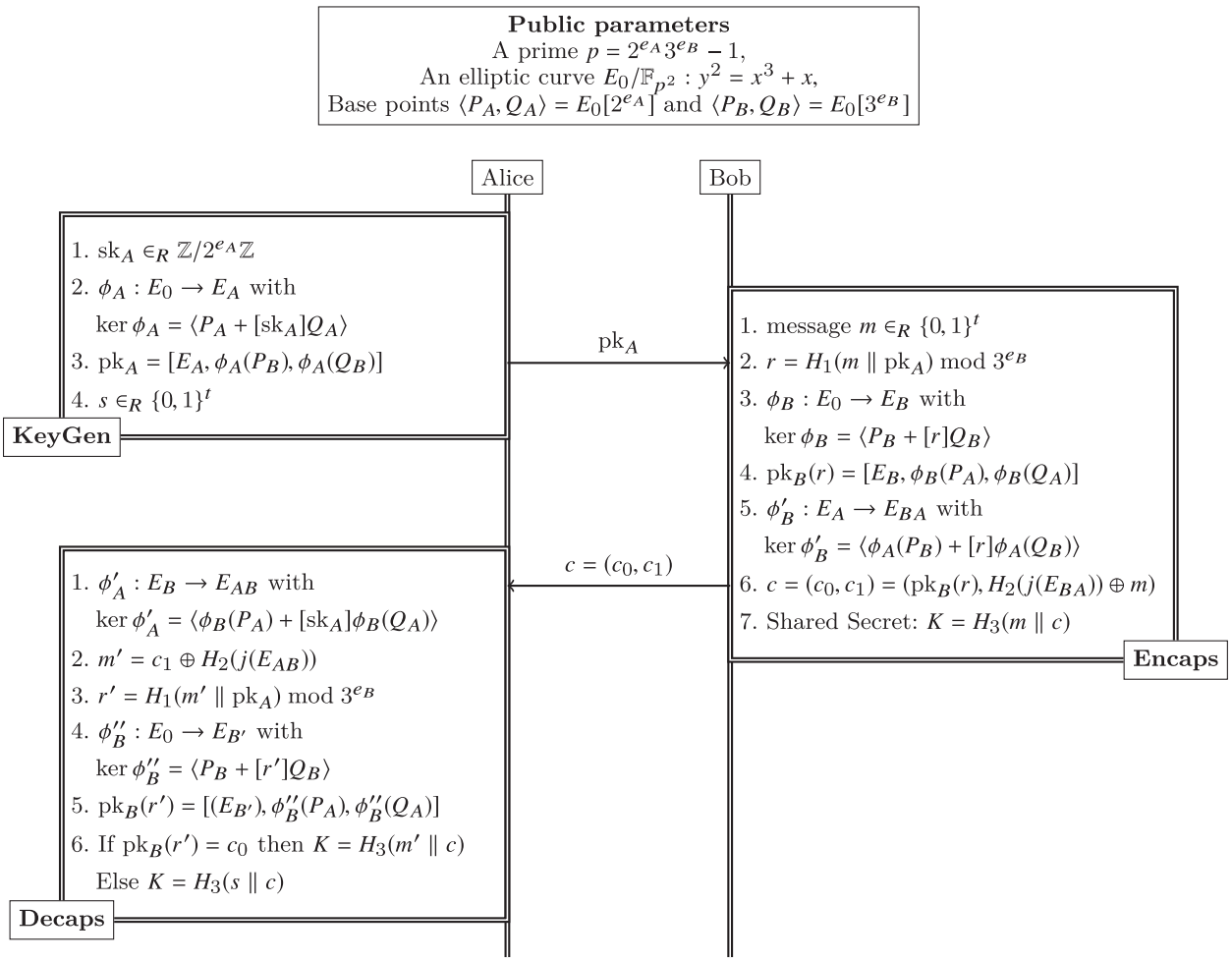

Fig. 3. SIKE mechanism.

degree 2 and 3 in [22], improved the SIKE's key compression overhead significantly compared to the previous works, while the public-key and ciphertext size are reduced by 59%.

SIKE's key compression feature offers a flexibility in protocol design for key encapsulation mechanism in different environments: the required bandwidth can be shrunk down almost in half if the performance is not a bottleneck.

Recall from SIDH key exchange that Alice's public key is $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ and similarly Bob's public key is $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$. We can shortly represent those public keys $\{E, \phi(P), \phi(Q)\}$. That is, the public key consists of an elliptic curve over $\mathbb{F}_{p^2}$ and two points over $\mathbb{F}_{p^2}$ on this curve. However, as the public keys are similar in SIKE mechanism, they are encoded by x-coordinates of three points (see [4] for details):

$$\{X_{\phi(P)}, X_{\phi(Q)}, X_{\phi(P-Q)}\}$$

of size $6 \log p$ in total. Note that this encoding can be deterministically converted to original public key.

It is further improved by Azarderakhsh *et al.* [6], by sending in the form

$$\{j(E) \in \mathbb{F}_{p^2} \text{ and } a_1, a_2, b_1, b_2 \in \mathbb{Z}_{3^n}\},$$

so that $\phi(P) = a_1\ R_1 + a_2\ R_2$ and $\phi(Q) = b_1\ R_1 + b_2\ R_2$ for some pre-shared canonical basis $\langle R_1, R_2\rangle = 3^n$. In that case, it reduces $4 \log p$ total key size bits but 10 times slower compression/decompression. Moreover, Costello *et al.* [11] reduces the total key size $3.5 \log p$ bits but 2.4 times slower compression/decompression by sending in the form

$$\{j(E) \in \mathbb{F}_{p^2} \text{ and } \alpha, \beta, \gamma \in \mathbb{Z}_{3^n}\},$$

so that $\alpha = b_1\ a_1^{-1}, \beta = a_2\ a_1^{-1}, \gamma = b_2\ a_1^{-1}$. This compression methods are also given in Figure 4. Together with the works of Zanon *et al.* [26] and Naehrig and Renes [22], compression/decompression running time is reduced significantly. The running times for actual operation required for key compression can be seen in Table 2 of [22]. In this paper, we used the latest library SIDHv3.2 which includes the latest improvement by [22].

III. Target 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

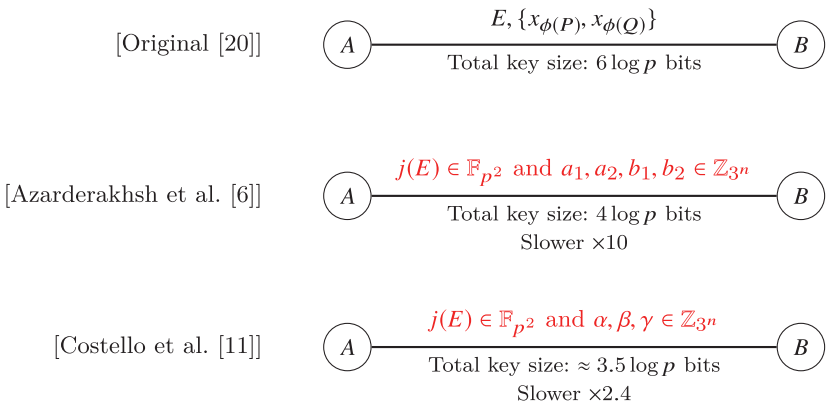

Fig. 4. Key compression.

TABLE I
COMPARISON OF PUBLIC KEY SIZE BETWEEN STANDARD AND COMPRESSED VERSIONS

NIST Security Level	Prime	Public Key Size (in bytes)	
		Standard	Compressed
Level I	SIKEp434	330	196
Level II	SIKEp503	378	224
Level III	SIKEp610	462	273
Level V	SIKEp751	564	331

TABLE II
COMPARISON OF MULTIPLICATION METHODS, IN TERMS OF THE NUMBER OF ADDITION OPERATIONS DEPENDING ON THE NUMBER OF WORD

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

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. Apple 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` ← (`X1` × `X2`) mod 2^{64}.
- `UMULH` (unsigned multiplication, high part): `UMULH X0, X1, X2` computes `X0` ← (`X1` × `X2`)/2^{64}.

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

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

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

IV. OPTIMIZED FIELD ARITHMETIC IMPLEMENTATION

There are 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 [18], [19], [24]. 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 [10]. In particular, they used 2-level Karatsuba for 964-bit finite field multiplication, which shows 23.9% performance enhancements than conventional Comba method. In [24], 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 pipeline stalls. In [17], Jalali et al. presented the optimized Montgomery multiplication by mixing AArch64 and ASIMD instructions. This approach shows the better performance than Comba method, when the operand size is long enough, such as SIKEp751 and SIKEp964.

Recently, two novel SIKE protocols (i.e. SIKEp434 and SIKEp610) for NIST Post Quantum Cryptography competition round 2 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 order to achieve high performance, the arithmetic for SIKEp434 and SIKEp610 is optimized to utilize the ARMv8-A ability fully. Furthermore, we also include the performance of compressed SIKEp434 and compressed SIKEp610. To describe the multi-precision arithmetic, we used following notations. Let A and B be operands of length m bits each. Each operand is written as $A = (A[n-1], \ldots, A[1], A[0])$ and $B = (B[n-1], \ldots, B[1], B[0])$, where $n = \lceil m/w \rceil$ is the number of words to represent operands, m is operand length, and w is the computer word size (i.e. 64-bit). The addition result ($C = A + B$) is represented as $C = (C[n-1], \ldots, C[1], C[0])$. For the multiplication ($C = A \times B$), the result is represented as $C = (C[2n-1], \ldots, C[1], C[0])$.

A. Finite Field Addition and Subtraction

Finite field addition and subtraction operations firstly need to perform addition and subtraction operations, respectively (See Section 2.2.1 of [15] for details). Afterward, intermediate results are reduced through the reduction routine, when the carry or borrow bit is detected. In order to avoid the timing attack, the reduction routine is performed without conditional statements (i.e. constant timing). To achieve this property, we used the masked modular reduction approach, which always perform regular routines, regardless of the carry or borrow bit (See Section 4.4 of [23] for details). When the carry or borrow bit is detected, the mask value in word is set to $2^{64}-1$. Otherwise, the mask value is set to 0. With the mask value, the modulus is determined, whether it is modulus value or 0.

For the 434-bit addition or subtraction operation, we utilized 14 general purpose registers to store the operands (i.e. $2 \times \lceil 434/64 \rceil$) since each operand requires 7 registers. In particular, two words of 434-bit modulus are $2^{64}-1$ (i.e. `0xFFFFFFFFFFFFFFFF`). We only set one word to $2^{64}-1$ and use this twice for computations, which reduces one operand setting overheads. For 610-bit addition or subtraction operation, we utilized 20 general purpose registers to retain all operands (i.e. $2 \times \lceil 610/64 \rceil$) since each operand requires 10 registers. Similarly, three words of 610-bit modulus are set to $2^{64}-1$ (i.e. `0xFFFFFFFFFFFFFFFF`). This word is used three times with only one memory access, which reduces two operand setting overheads.

B. Multiplication

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 intermediate results. In each MAC routine, two multiplication (`MUL_LOW` and `MUL_HIGH`) and three addition

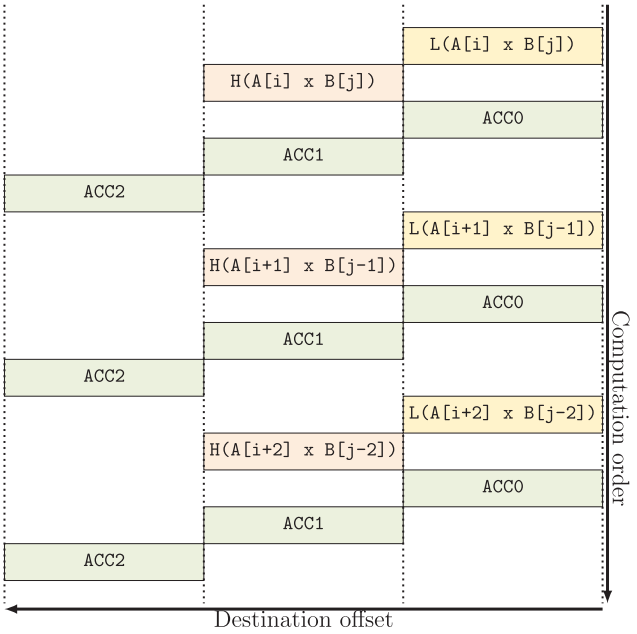

Fig. 5. Part of column-wise multiplication for ARMv8, where L, H, and ACC represent lower multiplication, higher multiplication, and accumulation, respectively.

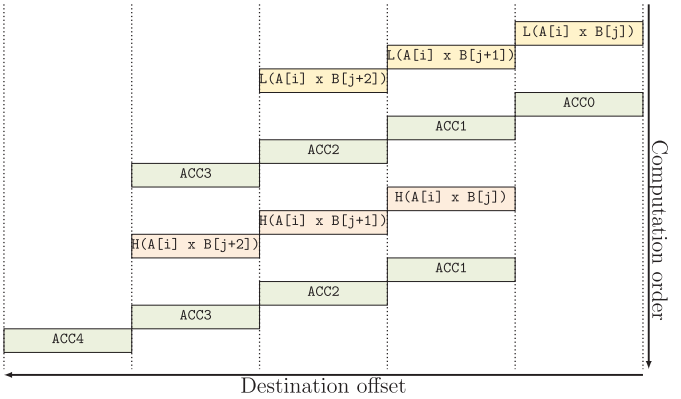

Fig. 6. Part of row-wise multiplication for ARMv8, where L, H, and ACC represent lower multiplication, higher multiplication, and accumulation, respectively.

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

In this work, we target the relatively shorter modulus (i.e. 434-bit) than previous works (i.e. 751-bit or 964-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 $\lceil m/w \rceil$, operand length, and word size, respectively. Under the 64-bit processor setting, the n is set to 7 for 434-bit ($\lceil 434/64 \rceil$). Considering that ARMv8 supports 31 64-bit registers, the required number of registers for 434-bit can be retained in registers. In Figure 6, the part of MAC routine in row-wise method for 64-bit ARMv8 processors is described. The example performs 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 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-word 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 word are given in Table II. 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-word case (i.e. 434-bit), the row-wise method reduces 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-word case but it generates a number of sub-routines to perform the computation and store the intermediate results, which requires additional operations and memory accesses [21].

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 operands, we also need registers for intermediate results and temporal storage. Due to limited number of registers, we only maintain half number of operands in registers and load remaining operands on demand.

Karatsuba's method reduces a multiplication of two m-bit operands to three $\frac{m}{2}$-bit multiplications with some addition or subtraction operations. There are two approaches to perform Karatsuba multiplication, including additive Karatsuba and subtractive Karatsuba. Taking the multiplication of n-word operand A and B as an example, operands are represented as $A = A_H \cdot 2^{\frac{n}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{n}{2}} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the following equation by using additive Karatsuba's method:

$$A_H \cdot B_H \cdot 2^n + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{n}{2}} + A_L \cdot B_L \quad (1)$$

or subtractive Karatsuba's method:

$$A_H \cdot B_H \cdot 2^n + [A_H \cdot B_H + A_L \cdot B_L - |A_H - A_L| \cdot |B_H - B_L|] \cdot 2^{\frac{n}{2}} + A_L \cdot B_L \quad (2)$$

For the 610-bit multiplication, 1-level subtractive Karatsuba multiplication is used, which consists of 3 320-bit multiplication operations with some addition or subtraction operations.

In the beginning, we 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
  ⋮
```

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 cost.

C. Reduction

In this section, we adapt techniques described in previous sections to implement modular multiplication for SIKE. Specifically, we target parameter sets based on SIKEp434 and SIKEp610 [4].

Multi-precision modular multiplication is the most expensive operation for the implementation of SIKE [12], [20]. 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 [24].

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 of modulus is set to 0. To reduce memory accesses, we keep as many results as possible in registers. Since Montgomery multiplication performs partial products with modulus and quotient (The quotient is intermediate results multiplied by constant m'), we maintained all quotients in 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, where n-word computation. For instance, 16 and 22 memory accesses for SIKEp434 and SIKEp610 are optimized.

1) Interleaved Montgomery Multiplication: The Algorithm 1 describes Montgomery multiplication. The Montgomery multiplication firstly performs the operand multiplica tion. Second, the quotient (Q) is generated by multiplying the

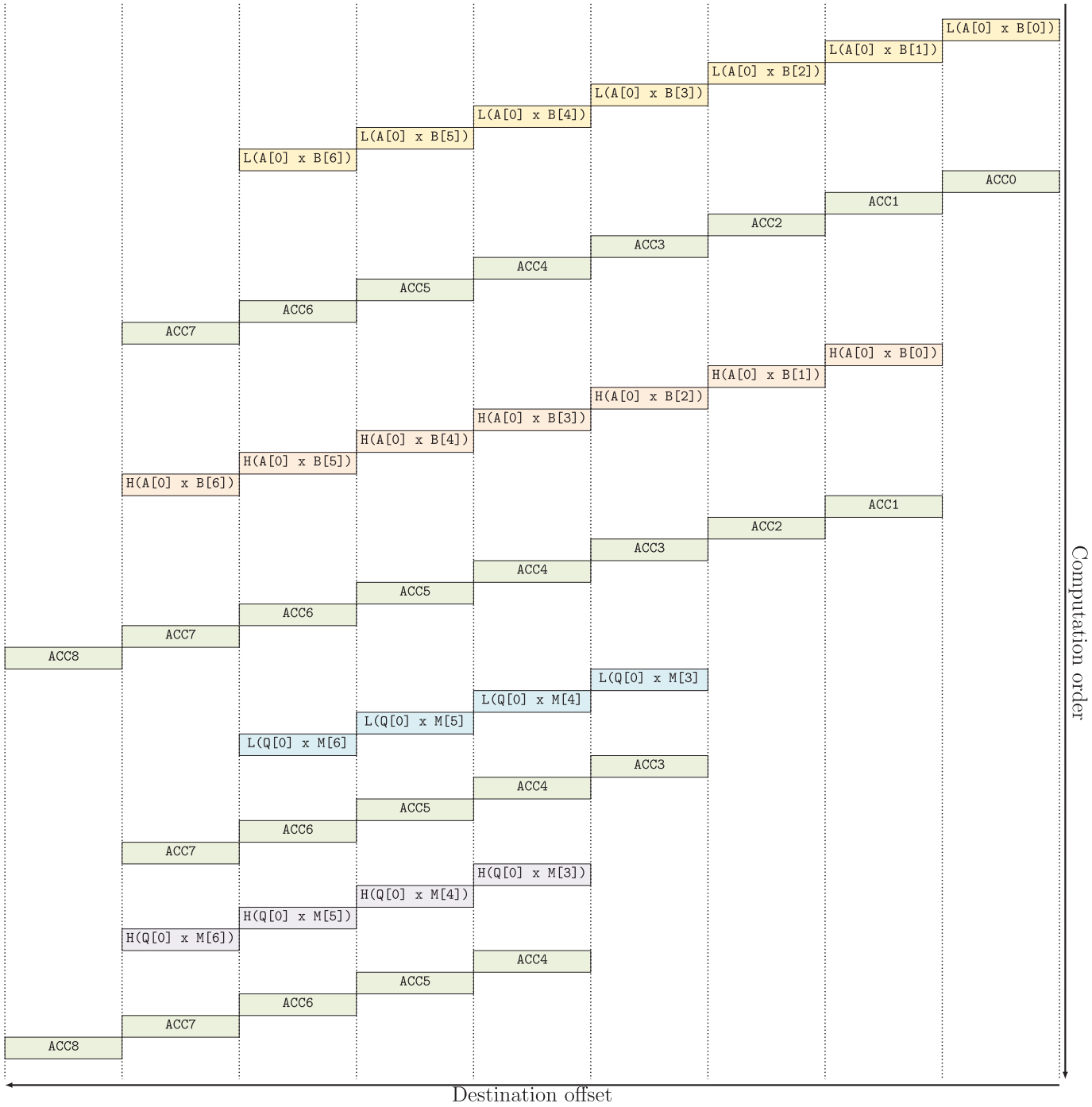

Fig. 7. Part of interleaved Montgomery multiplication for SIKEp434, where L, H, and ACC represent lower multiplication, higher multiplication, and accumulation, respectively.

intermediate result (T) and modulus inverse constant (M') Third, the quotient (Q) is multiplied by the modulus (M) and the result is added to the intermediate result (T).

The separated Montgomery multiplication performs the multiplication (Step 1) and reduction (Step 2 and Step 3) in separated way. The alternative way to implement the Montgomery multiplication is interleaved way. The part of multiplication is performed and the intermediate result is directly reduced. Compared with the separated approach, this method optimizes the number of memory accesses for intermediate result loading and storing. Total $2n$ memory accesses are optimized (n for memory loading and n for memory storing). We utilized the interleaved Montgomery multiplication for SIKEp434, since all operands and intermediate results are well retained in general purpose registers. ARMv8 processors support 31 general purpose registers and we utilized 30 registers. In particular, 4, 1, 7, 10, and 8 registers are allocated for modulus, operand A, operand B, intermediate result, and temporal storage, respectively. The part of interleaved Montgomery multiplication for SIKEp434 is given in Figure 7.

TABLE III

COMPARISON OF IMPLEMENTATIONS OF THE SIKEP434 AND SIKEP610 ARITHMETIC ON ARMV8 RMV8 CORTEX-A55@1.766GHZ AND CORTEX-A75@2.803GHZ BASED PROCESSOR. TIMINGS ARE REPORTED IN TERMS OF CLOCK CYCLES

Implementation	Language	Processor	Protocol	Timings [*cc*] (Performance Improvement)			
				$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ inv
SIKE (SIDH-v3.2) [13]	C	Cortex-A55	SIKEp434	171	127	2976	1572510
This work	ASM			74 (56%)	67 (47%)	602 (79%)	340425 (78%)
SIKE (SIDH-v3.2) [13]	C	Cortex-A75		272	202	4723	2495892
This work	ASM			34 (87%)	28 (86%)	561 (88%)	309359 (87%)
SIKE (SIDH-v3.2) [13]	C	Cortex-A55	SIKEp610	249	182	6093	4469981
This work	ASM			122 (51%)	111 (39%)	1287 (78%)	955305 (78%)
SIKE (SIDH-v3.2) [13]	C	Cortex-A75		224	154	5673	4156698
This work	ASM			50 (77%)	42 (72%)	1046 (81%)	777992 (81%)

Algorithm 1 Calculation of the Montgomery Multiplication

Require: An odd m-bit modulus M, Montgomery radix $R = 2^m$, an operand T where operands (A and B in the range $[0, m-1]$), and pre-computed constant $M' = -M^{-1} \bmod R$

Ensure: Montgomery product $Z = \text{MonMul}(A, B, R) = (A \times B) \cdot R^{-1} \bmod M$

1: $T \leftarrow A \cdot B$
2: $Q \leftarrow T \cdot M' \bmod R$
3: $Z \leftarrow (T + Q \cdot M)/R$
4: **return** Z

The first word of operand A is multiplied by all operand B ($B[0] \sim B[6]$) and the intermediate result is accumulated. Second, the first word of quotient Q is multiplied by modulus and the intermediate result is accumulated. With this approach, we can optimize the memory access to the intermediate result.

For SIKEp610 implementation, we used the separated Montgomery multiplication since the required number of registers is over available registers, which introduces a number of memory accesses.

2) Shifted Modulus: The execution timing of modular reduction is relied on the number of partial products. For this reason, reducing the number of partial products can lead to the performance enhancements. The general form of modulus for SIKEp610 is as follows.

```
      Normal representation
M[4]  0x6E02000000000000
M[5]  0xB1784DE8AA5AB02E
M[6]  0x9AE7BF45048FF9AB
M[7]  0xB255B2FA10C4252A
M[8]  0x819010C251E7D88C
M[9]  0x000000027BF6A768
```

This requires 6 registers to retain all modulus. The number of required registers can be optimized with the shifted representation. The least significant word of modulus ($M[4]$) only utilize the 15-bit out of 64-bit and the most significant word of modulus ($M[9]$) has 30-bit empty space. We shifted the modulus by 16-bit to the left and the representation is re-written as follows.

```
      Shifted representation
M[4]  0x4DE8AA5AB02E6E02
M[5]  0xBF45048FF9ABB178
M[6]  0xB2FA10C4252A9AE7
M[7]  0x10C251E7D88CB255
M[8]  0x00027BF6A7688190
```

The shifted representation only requires 5 registers for SIKEp610 modulus. This optimized the number of partial products by 10 ($60 \rightarrow 50$). The shifted computation utilizes the shifted representation. The result is obtained in two steps. First, the ordinary multiplication is performed. Second, the intermediate result is shifted. For SIKEp610, we shifted 16-bit to the left and the result should be reordered after computations. The reorder computation is described as follows.

```
⋮
LSL  X7,X22,#48
ORR  X7,X7,X21,LSR#16
⋮
```

The higher word (`X22`) is shifted to the left by 48-bit and the lower word (`X21`) is shifted to the right by 16-bit. To optimize the shift and addition operation, we utilized the barrel-shifter module, which can perform the shift operation on the second operand without additional costs.

For the SIKEp434 case, the modulus $(p+1)$ is divided into 4-word. The shifted modulus needs to ensure that the summation of remaining bits of the least significant word and the most significant word is over 64-bit. However, the SIKEp434 case only has 39-bit (14-bit from the most significant word and 25-bit from the least significant word). The following is the modulus of SIKEp434.

```
M[3]  0xFDC1767AE3000000
M[4]  0x7BC65C783158AEA3
M[5]  0x6CFC5FD681C52056
M[6]  0x0002341F27177344
```

TABLE IV

COMPARISON OF IMPLEMENTATIONS OF THE SIKE PROTOCOLS ON ARMV8 CORTEX-A55@1.766GHZ AND CORTEX-A75@2.803GHZ BASED PROCESSORS. TIMINGS ARE REPORTED IN TERMS OF MILLISECONDS AND CLOCK CYCLES. ^{m}: MIXED APPROACH

Processor	Protocol		Implementation	Language	Timings [milliseconds]				Timings [$cc \times 10^6$]			
					KeyGen	Encaps	Decaps	Total	KeyGen	Encaps	Decaps	Total
Cortex-A55	Standard	SIKEp434	SIKE (SIDH-v3.2) [13]	C	62.2	101.7	108.5	210.2	109.8	179.5	191.6	371.2
			This work	ASM	15.4	25.3	27.0	52.4	27.2	44.8	47.7	92.5
		SIKEp503	SIKE (SIDH-v3.2) [13]	C	95.5	157.2	167.4	324.6	168.6	277.6	295.7	573.2
			SIKE (SIDH-v3.2) [13]	ASM	21.9	35.3	38.0	73.4	38.6	62.4	67.2	129.6
			Jalali et al. [17]	ASM	27.6	45.1	48.2	93.4	48.7	79.7	85.2	164.9
			Jalali et al. [17]^m	ASM	39.6	64.8	69.2	134.0	69.9	114.4	122.3	236.7
		SIKEp610	SIKE (SIDH-v3.2) [13]	C	181.4	333.7	335.6	669.3	320.4	589.2	592.7	1181.9
			This work	ASM	43.5	79.9	80.7	160.6	76.8	141.2	142.4	283.6
		SIKEp751	SIKE (SIDH-v3.2) [13]	C	328.3	532.2	571.7	1103.9	579.8	939.9	1009.7	1949.6
			SIKE (SIDH-v3.2) [13]	ASM	74.4	118.3	128.5	246.8	131.3	208.9	226.9	435.8
			Jalali et al. [17]	ASM	85.3	137.9	148.5	286.3	150.6	243.5	262.2	505.6
			Jalali et al. [17]^m	ASM	102.9	165.7	178.8	344.5	181.7	292.7	315.7	608.3
	Compressed	SIKEp434	SIKE (SIDH-v3.2) [13], [22]	C	102.6	159.0	152.0	311.0	181.2	280.8	268.5	549.3
			This work [22]	ASM	25.2	39.3	37.0	76.3	44.5	69.3	65.3	134.7
		SIKEp503	SIKE (SIDH-v3.2) [13], [22]	C	156.6	242.3	228.1	470.4	276.5	427.9	402.8	830.8
			SIKE (SIDH-v3.2) [13], [22]	ASM	35.9	56.3	52.8	109.1	63.4	99.4	93.3	192.7
		SIKEp610	SIKE (SIDH-v3.2) [13], [22]	C	314.4	461.8	451.5	913.3	555.2	815.6	797.3	1612.9
			This work [22]	ASM	74.0	110.1	108.2	218.4	130.7	194.5	191.1	385.6
		SIKEp751	SIKE (SIDH-v3.2) [13], [22]	C	528.9	834.2	780.9	1615.1	934.0	1473.1	1379.1	2852.2
			SIKE (SIDH-v3.2) [13], [22]	ASM	117.2	190.6	178.2	368.8	207.0	336.6	314.7	651.3
Cortex-A75	Standard	SIKEp434	SIKE (SIDH-v3.2) [13]	C	36.4	59.6	63.6	123.2	102.1	167.1	178.3	345.4
			This work	ASM	8.1	13.4	14.3	27.6	22.8	37.5	40.0	77.5
		SIKEp503	SIKE (SIDH-v3.2) [13]	C	56.4	92.9	98.7	191.6	158.0	260.3	276.8	537.1
			SIKE (SIDH-v3.2) [13]	ASM	9.6	15.8	16.9	32.7	27.0	44.3	47.3	91.6
			Jalali et al. [17]	ASM	11.6	19.1	20.3	39.4	32.5	53.5	57.0	110.5
			Jalali et al. [17]^m	ASM	14.2	23.2	24.8	48.1	39.8	65.2	69.6	134.8
		SIKEp610	SIKE (SIDH-v3.2) [13]	C	106.6	196.3	197.4	393.7	298.8	550.3	553.2	1103.5
			This work	ASM	21.0	38.8	39.0	77.8	58.9	108.7	109.3	218.0
		SIKEp751	SIKE (SIDH-v3.2) [13]	C	193.3	313.6	336.7	650.4	541.7	879.1	943.9	1823.0
			SIKE (SIDH-v3.2) [13]	ASM	32.3	52.3	56.3	108.6	90.6	146.5	157.8	304.3
			Jalali et al. [17]	ASM	38.6	62.6	67.3	130.0	108.3	175.6	188.8	364.4
			Jalali et al. [17]^m	ASM	38.7	62.5	67.3	129.9	108.4	175.2	188.8	364.0
	Compressed	SIKEp434	SIKE (SIDH-v3.2) [13], [22]	C	59.5	93.2	87.6	180.8	166.7	261.2	245.5	506.8
			This work [22]	ASM	13.3	20.7	19.8	40.6	37.3	58.1	55.6	113.7
		SIKEp503	SIKE (SIDH-v3.2) [13], [22]	C	91.9	142.7	137.3	280.0	257.7	400.1	384.7	784.8
			SIKE (SIDH-v3.2) [13], [22]	ASM	16.2	24.8	23.4	48.2	45.4	69.6	65.6	135.2
		SIKEp610	SIKE (SIDH-v3.2) [13], [22]	C	182.2	269.8	265.2	535.0	510.7	756.2	743.4	1499.5
			This work [22]	ASM	36.1	53.1	52.1	105.2	101.1	148.8	146.1	294.9
		SIKEp751	SIKE (SIDH-v3.2) [13], [22]	C	309.1	489.9	468.1	957.9	866.4	1373.1	1312.0	2685.1
			SIKE (SIDH-v3.2) [13], [22]	ASM	51.6	82.9	77.4	160.3	144.6	232.4	216.9	449.3

V. PERFORMANCE RESULT

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

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

Table III summarizes 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 based processors, we compare results with the SIKE round 2 reference code [13]. The *unoptimized* reference implementation is written in C using the SIKE round 2 library [4]. In this case, proposed arithmetic implementations show much higher performance than reference work. Finite field addition for SIKEp434 and SIKEp610 shows performance enhancements by 2.67x/8.77x and 2.39x/4.66x on Cortex-A55/Cortex-A75, respectively. Finite field subtraction for SIKEp434 and SIKEp610 shows performance enhancements by 2.28x/8.08x and 1.93x/3.94x

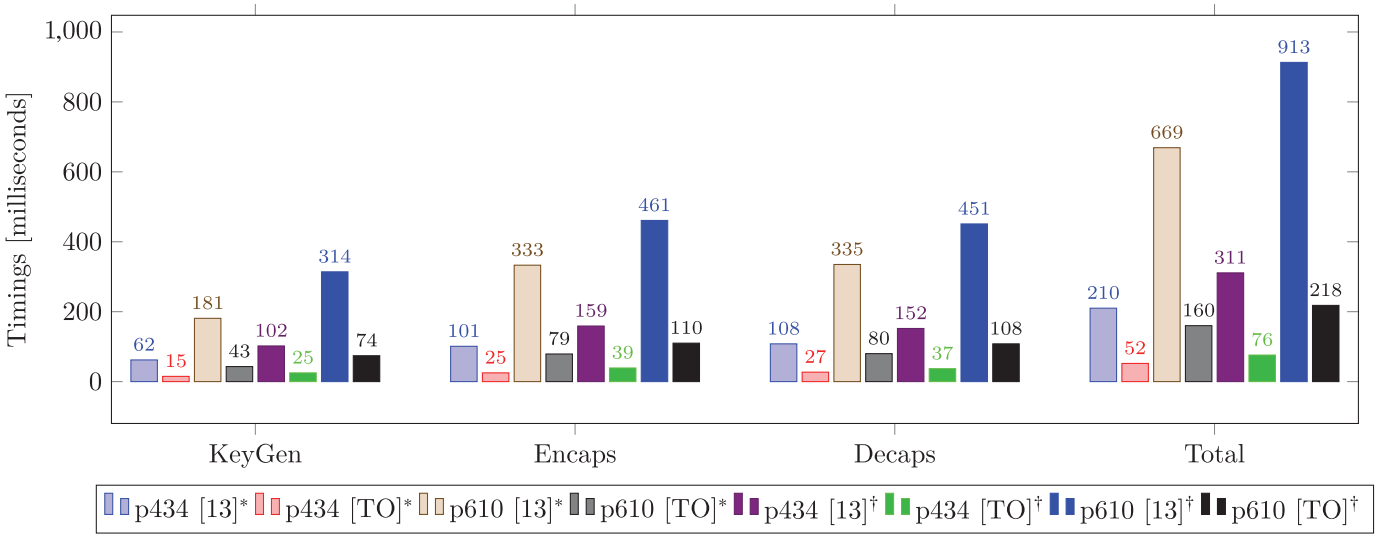

Fig. 8. Comparison of implementations of the SIKE protocols on ARMv8 Cortex-A55@1.766GHz based processors. Timings are reported in terms of milliseconds. *: standard version, †: compressed version, TW: this work.

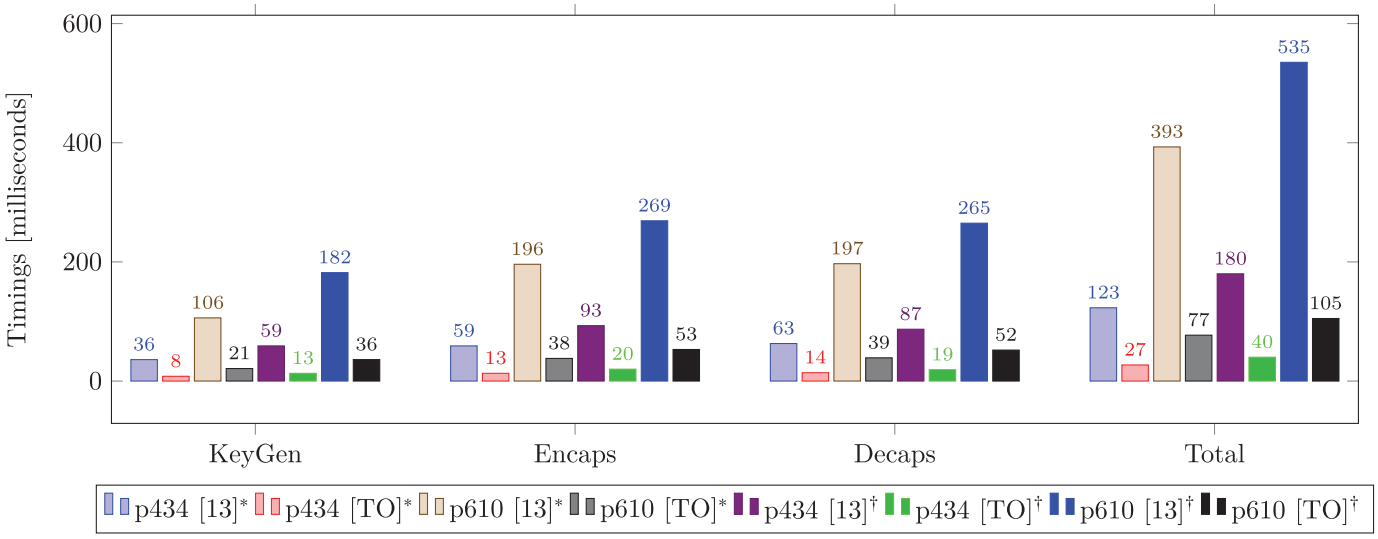

Fig. 9. Comparison of implementations of the SIKE protocols on ARMv8 Cortex-A75@2.803GHz based processors. Timings are reported in terms of milliseconds. *: standard version, †: compressed version, TW: this work.

on Cortex-A55/Cortex-A75, respectively. Finite field multiplication for SIKEp434 and SIKEp610 shows performance enhancements by 5.01x/8.41x and 4.76x/5.45x on Cortex-A55/Cortex-A75, respectively. Finite field inversion for SIKEp434 and SIKEp610 shows performance enhancements by 4.66x/8.07x and 4.70x/5.36x on Cortex-A55/Cortex-A75, respectively.

As described above, performance enhancements are observed in both ARMv8 processors but Cortex-A75 shows the better performance improvements than Cortex-A55. This difference comes from their different computer architectures. Cortex-A55 supports 2-wide decode in-order superscalar pipeline while Cortex-A75 supports 3-wide decode out-of-order superscalar pipeline. The assembly code is optimized further through out-of-order superscalar pipeline of Cortex-A75. Between SIKEp434 and SIKEp610, SIKEp434 is optimized further than SIKEp610 since the implementation of SIKEp434 can take advantage of the optimal register usage due to the short operand.

Table IV summarizes results of different software implementations of the SIKEp434 and SIKEp610 protocols on ARMv8 Cortex-A55 and Cortex-A75 processors. For standard SIKE protocols, proposed implementations of SIKEp434 and SIKEp610 outperform previous works by 3.76x/4.04x and 3.98x/4.57x for Cortex-A55/Cortex-A75 processors, respectively. Considering that target processors are working at 1.766GHZ and 2.803GHz, SIKEp434 and SIKEp610 require only 0.055/0.030 and 0.168/0.086 seconds for Cortex-A55/Cortex-A75 processors, respectively. Compared with other SIKE protocols, the SIKEp434 shows the highest performance and the SIKEp751 shows the lowest performance. We also evaluated the key compressed SIKE protocols.

For compressed SIKE protocols, proposed implementations of SIKEp434 and SIKEp610 outperform previous works by 4.07x/4.45x and 4.18x/5.08x for Cortex-A55/Cortex-A75 processors, respectively. Key compressed SIKEp434 and SIKEp610 require only 0.076/0.040 and 0.218/0.105 seconds for Cortex-A55@1.766GHz/Cortex-A75@2.803GHz processors, respectively. Compressed versions require more timing than standard SIKE protocols but the execution timing is reasonably fast enough for real world applications.

The performance comparison is given in Figure 8 and 9 for Cortex-A55 and Cortex-A75, respectively. The fastest performance of standard version is obtained from proposed implementation of SIKEp434. For the compressed version, proposed SIKEp434 implementation also achieved the fastest performance.

Overall, the proposed assembly implementation achieved significant performance improvements. The assembly implementation has several advantages over C based implementation. First, the customized register utilization is available. By carefully designing variable assignments, many variables are kept in registers, which reduces the number of memory accesses. Second, the assembly implementation can handle status registers. This avoids a number of carry handling routines in C language. Third, C implementation is mainly relied on compiler's capability. If the compiler selects inefficient instructions, this leads to slow performance.

VI. Conclusion

This paper presented high-speed implementation of SIKE round 2 on high-end 64-bit ARMv8 Cortex-A55 and Cortex-A75 processors. A combination of several optimization methods yields very efficient modular multiplications for SIKEp434 and SIKEp610 protocols that are shown, for example, to be approximately 5.01x/8.41x and 4.76x/5.45x faster than the normal modular multiplication implementations for "SIDH-friendly" modulus on a 64-bit ARMv8 Cortex-A55/Cortex-A75 processors. The optimized implementations which push further the performance of post-quantum supersingular isogeny-based protocols, are 3.98x/4.57 faster than the previous implementations of SIKEp610, targeting the Cortex-A55/Cortex-A75 processors. Furthermore, we integrated our fast modular arithmetic implementations, compact prime SIKEp434, and optimal strategy for isogeny computations into Microsoft's SIDH library. A 128-bit full key-exchange execution over optimal prime SIKEp434 is performed in about 0.055/0.030 seconds on a ARMv8 Cortex-A55@1.766GHz/Cortex-A75@2.803GHz processors, which show the practicality of isogeny based post-quantum cryptography over mobile devices. The key compressed versions are also evaluated and SIKEp434 is performed in about 0.076/0.040 seconds on a ARMv8 Cortex-A55@1.766GHz/Cortex-A75@2.803GHz processors.

Inspired by recent IETF draft [1] on SIKE integration into Amazon's AWS services, supporting hybrid post-quantum KEM on one of the largest cloud providers in industry, this work improves the overall performance of the key encapsulation mechanism significantly and provides guidelines for industry practitioners to enhance SIKE's performance on the high-performance ARM Cortex-A processors which are used vastly in the new generation of cellphones.

References

[1] (2019). *BIKE and SIKE Hybrid Key Exchange Cipher Suites for Transport Layer Security (TLS)*. Accessed: Nov. 2019. [Online]. Available: https://tools.ietf.org/html/draft-campagna-tls-bike-sike-hybrid-00

[2] 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," in *Proc. 25th Int. Conf. Sel. Areas Cryptogr. (SAC)*, 2018, pp. 322–343.

[3] ARM Limited. (2017). *ARM Architecture Reference Manual ARMv8, for ARMv8-A Architecture Profile*. [Online]. Available: https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

[4] R. Azarderakhsh *et al.* (Feb. 2019). *Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process*. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions/SIKE.zip

[5] R. Azarderakhsh *et al.* (2017). *Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process*. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip

[6] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proc. 3rd ACM Int. Workshop ASIA Public-Key Cryptogr. (AsiaPKC)*, 2016, pp. 1–10.

[7] J. W. Bos and S. Friedberger, "Fast arithmetic modulo $2^x p^y \pm 1$," in *Proc. IEEE Symp. Comput. Arithmetic (ARITH)*, Jul. 2017, pp. 148–155.

[8] D. X. Charles, K. E. Lauter, and E. Z. Goren, "Cryptographic hash functions from expander graphs," *J. Cryptol.*, vol. 22, no. 1, pp. 93–113, Jan. 2009.

[9] L. Chen *et al.*, "Report post-quantum cryptography," U.S. Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. NISTIR 8105, 2016.

[10] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Syst. J.*, vol. 29, no. 4, pp. 526–538, 1990.

[11] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient compression of SIDH public keys," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Paris, France: Springer, 2017, pp. 679–706.

[12] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie-Hellman," in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 9814, M. Robshaw and J. Katz, Eds. Santa Barbara, CA, USA: Springer, 2016, pp. 572–601.

[13] C. Costello, P. Longa, and M. Naehrig. (2019). *SIDH Library*. [Online]. Available: https://github.com/Microsoft/PQCrypto-SIDH

[14] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the security of supersingular isogeny cryptosystems," in *Proc. 22nd Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2016, pp. 63–91.

[15] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer, 2006.

[16] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation," in *Proc. 15th Int. Theory Cryptogr. Conf. (TCC)* 2017, pp. 341–371.

[17] A. Jalali, R. Azarderakhsh, M. M. Kermani, M. Campagna, and D. Jao, "ARMv8 SIKE: Optimized supersingular isogeny key encapsulation on ARMv8 processors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 11, pp. 4209–4218, Nov. 2019.

[18] A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 5, pp. 902–912, Sep. 2019.

[19] A. Jalali, R. Azarderakhsh, and M. Mozaffari-Kermani, "Efficient post-quantum undeniable signature on 64-bit ARM," in *Proc. Int. Conf. Sel. Areas Cryptogr.* Ottawa, ON, Canada: Springer, 2017, pp. 281–298.

[20] D. Jao and L. D. Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Post-Quantum Cryptography* (Lecture Notes in Computer Science), vol. 7071, B. Yang, Ed. Taipei, Taiwan: Springer, 2011, pp. 19–34.

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

[22] M. Naehrig and J. Renes, "Dual isogenies and their application to public-key compression for isogeny-based cryptography," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 499, Dec. 2019.

[23] H. Seo, A. Jalali, and R. Azarderakhsh, "SIKE round 2 speed record on ARM Cortex-M4," in *Proc. Int. Conf. Cryptol. Netw. Secur.* Fuzhou, China: Springer, 2019, pp. 39–60.
[24] H. Seo, Z. Liu, P. Longa, and Z. Hu, "SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, 2018, pp. 1–20.
[25] The National Institute of Standards and Technology. (2018). *Post-Quantum Cryptography Standardization*. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization
[26] G. H. M. Zanon, M. A. Simplicio, G. C. C. F. Pereira, J. Doliskani, and P. S. L. M. Barreto, "Faster key compression for isogeny-based cryptosystems," *IEEE Trans. Comput.*, vol. 68, no. 5, pp. 688–701, May 2019.

Hwajeong Seo received the B.S.E.E., M.S., and Ph.D. degrees in computer engineering from Pusan National University. He is currently an Assistant Professor with Hansung University. His research interests include the Internet of Things and information security.

Pakize Sanal received the B.E. and M.E. degrees in computer engineering from Yasar University, Turkey, in 2016 and 2019, respectively. She is currently pursuing the Ph.D. degree in computer engineering with Florida Atlantic University under the supervision of Dr. Azarderakhsh. She is also a Research Assistant with I-SENSE Lab. Her research interests include efficient implementation of cryptographic algorithms and post-quantum cryptography.

Amir Jalali received the Ph.D. degree in computer engineering from the Department of Computer, Electrical Engineering and Computer Science, Florida Atlantic University, USA, in 2018. He is currently with the Information Security Group, LinkedIn Corporation. His current research interests include applied cryptography, post-quantum cryptography, and homomorphic encryption.

Reza Azarderakhsh (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Western University, in 2011. He was a recipient of the NSERC Post-Doctoral Research Fellowship working with the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Florida Atlantic University. His current research interests include finite field and its applications, elliptic curve cryptography, pairing-based cryptography, and post-quantum cryptography. He is serving as an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS.