Optimized Supersingular Isogeny Key Encapsulation on ARMv8 Processors

Amir Jalali[1], Reza Azarderakhsh[1], Mehran Mozaffari Kermani[2], Matthew Campagna[3], and David Jao[4]

[1] Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, FL, USA,
{ajalali2016, razarderakhsh}@fau.edu
[2] Department of Computer Science and Engineering, University of South Florida, FL, USA,
mehran2@usf.edu
[3] Amazon Web Services Inc., Seattle, WA, USA,
campagna@amazon.com
[4] Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada,
djao@uwaterloo.ca

Abstract. In this work, we present highly-optimized constant-time software libraries for Supersingular Isogeny Key Encapsulation (SIKE) protocol on ARMv8 processors. Our optimized hand-crafted assembly libraries provide the most efficient timing results on 64-bit ARM-powered devices. Moreover, the presented libraries can be integrated into any other cryptography primitives targeting the same finite field size. We design a new mixed implementation of field arithmetic on 64-bit ARM processors by exploiting the A64 and Advanced SIMD processing units working in parallel. Using these techniques, we are able to improve the performance of the entire protocol by the factor of 5× compared to optimized C implementations on 64-bit ARM high-performance cores, providing 83-, 124-, and 159-bit quantum-security levels. Furthermore, we compare the performance of our proposed library with the previous highly-optimized ARMv8 assembly library available in the literature. The implementation results illustrate the overall 10% performance improvement in comparison with previous work, highlighting the benefit of using mixed implementation over relatively-large finite field size.

Keywords: ARM assembly, finite field, isogeny-based cryptosystems, key encapsulation mechanism, post-quantum cryptography.

1 Introduction

In recent years, extensive amount of research has been devoted to quantum computers. These machines are envisioned to be able to solve mathematical problems which are currently unsolvable for conventional computers, because of their exceptional computational power from quantum mechanics. Therefore, if quantum computers are ever built in large scale, they will certainly be able to break many or almost all of the currently in-use public-key cryptosystems, the threat of which would be catastrophic to the confidentiality and integrity of any secure communication. To counteract this problem, post-quantum cryptography protocols are required to preserve the security in the presence of quantum adversaries. Regardless of whether we can estimate the exact time for the advent of the quantum computing era, we must begin to prepare the security protocols to be resistant against potentially-malicious power of quantum computing. Accordingly, NIST initiated a process to evaluate, and standardize one or more post-quantum public-key cryptography primitives [21]. Recently, the first round of submission of the post-quantum primitives is completed and all the proposals are publicly available[5] to evaluate in terms of the proof of security and efficiency.

The submitted public-key post-quantum cryptography (PQC) proposals are based on five different hard problems and they are categorized as code-based cryptography [24], (ring) lattice-based cryptography [13,1], hash-based cryptography [25], multivariate cryptography [23], and isogeny-based cryptography [17]. The isogeny-based cryptography is based on the hardness of computing the isogenies between two isomorphic elliptic curves and it provides a complete key encapsulation protocol. The proposed method is denoted as Supersingular Isogeny Key Encapsulation (SIKE) [16], and constructed upon the initial Diffie-Hellman key-exchange scheme proposed by Jao and De Feo [17].

SIKE protocol provides a standard method of key-exchange between two parties, and it has been claimed to be secure against large-scale quantum adversaries running the Shor's quantum algorithm [29]. Compared to other post-quantum candidates, supersingular isogeny problem is a much younger

[5] NIST Standardization Process (Accessed Feb. 2019):
`https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`

scheme and its security and performance need to be investigated more. In terms of performance, SIKE is not a fast protocol due to the extensive number of point arithmetic which are required for computing large-degree isogenies. However, because of its significant smaller size of secret-key and public-key compared to other PQC candidates, SIKE is a suitable option for the applications where communication bandwith is critical. Furthermore, since it is the only post-quantum cryptography protocol which is constructed on elliptic curves, hybrid cryptography protocols can be derieved from SIKE and classical elliptic curve cryptography (ECC) to make the transition towards post-quantum cryptography more convenient and practical.

The initial idea of constructing cryptography schemes from the isogenies of regular elliptic curves was introduced by Rostovtsev and Stolbunov [26] in 2006. Later, Charles-Lauter-Goren [8] presented a set of cryptography hash functions constructed from Ramanujan graphs, i.e., the set of supersingular elliptic curves over $\mathbb{F}_{p^2}$ with ℓ-isogenies. The main breakthrough in constructing a post-quantum cryptography protocol based on the hardness of computing isogenies was proposed by Jao and De Feo in 2011 [17]. Their proposed scheme presents a set of public-key cryptography schemes such as key-exchange and encryption-decryption protocols with a coherent proof of security. Later in 2014, De Feo et al. [10] presented the first practical implementation of the Supersingular Isogeny Diffie-Hellman (SIDH) key-exchange protocol using optimized curve arithmetic techniques such as Montgomery arithmetic. Since the introduction of supersingular isogeny public-key protocol, many different schemes and implementations such as digital signature [12,33], undeniable signature [18,14], and static-static key agreement [5] have been proposed which are all built on the hardness of computing isogenies. The fast hardware architectures for computing isogenies of elliptic curves proposed by Koziel et al. [19] demonstrated that isogeny-based crptography has the potential to be considered as a practical candidates on FPGAs. However, the initial performance evaluations in software were not promising compared to other post-quantum candidates. In particular to those which are constructed over learning with errors problem [7,3,6]. The SIDH projective formulas and implementation by Costello et al. [9] smashed the performance bar of the protocol considerably by eliminating field inversions in isogeny computations, and provided an optimized software for key-exchange protocol on Intel processors, taking advantage of hand-written assembly implementation of finite field arithmetic. However, their software still suffered from the large amount of computations due to the arithmetic of elliptic curves. On ARM-powered devices, the performance evaluations are even worse due to the reduced instruction set computing (RISC) which concentrates on power-efficiency rather than performance [20,15]. Recently, a new set of optimizations in computation of Montgomery ladder and field arithmetic operations [11] improved the performance of the isogeny-based key-exchange protocol further; however, more investigation on the efficiency of ARM-based implementations is still required.

In this work, we investigate different approaches for implementing highly-optimized arithmetic libraries on 64-bit ARM processors. We introduce a new way of implementing multi-precision multiplication using mixed A64/ASIMD hand-written assembly along with pure A64 assembly implementation. All the previous works on this area concentrated on development of highly-optimized implementation of field arithmetic using pure SIMD instructions [2,20,15], or investigated a combination of SIMD and general register implementation on 32-bit ARM platforms, taking advantage of a mixture of 128-bit wide NEON vectors and 32-bit general registers [22,28]. In this work, we show that such a combination still can be beneficial on 64-bit ARMv8 family of processors by adopting a novel engineering technique which takes advantage of out-of-order execution pipeline on high-performance ARM cores. We compare the performance of our arithmetic libraries inside the SIKE reference implementation, and conclude the benefits of using mixed implementation over relatively-large finite fields.

1.1 Contributions

In this work, we study different approaches of implementing SIKE on 64-bit ARM. We engineer the finite field arithmetic implementation accurately to provide the fastest timing records of the protocol on our target platforms. Our contributions can be categorized as follows:

- We propose a new approach for implementing finite field arithmetic on 64-bit ARM processors. We combine general register and vector limbs in an efficient way to reduce the pipeline stalls and improve the overall performance. To the best of our knowledge, this work is the first implementation of such a technique on ARMv8 processors.
- We implement different optimized versions of finite field multiplication using Karatsuba multiplication method which outperforms the previous implementation of the field multiplication with the same size on ARMv8 target platform. The proposed implementations are constant-time and resistant to timing attacks.
- Our optimized software provides a constant-time implementation of the post-quantum SIKE protocol over three different quantum security levels. We state that, this work is the first implementation of `SIKEp964` which provides 159-bit quantum security level.

– We provide a comprehensive timing results of the SIKE protocol, using different hand-written assembly techniques along with optimized C benchmarks on ARMv8 processors. We are able to improve the overall performance of the SIKE library significantly on target processors.

In Section 2 we recall the essential concepts from [17,16] which are required in SIKE. In Section 3 we describe our implementation techniques and methodology to develop a highly-optimized field multiplier on ARMv8 processors. In Section 4 we present the efficient implementation of key components inside SHA-3 inside the SIKE protocol. We present performance results of our implementation and the comparison with previous work in Section 5. We conclude the paper in Section 6.

1.2 Code Availability

For reproducibility, we provide our optimized implementation publicly available. The source codes are available at: `https://github.com/amirjalali65/armv8-sike`.

2 Background

This section provides a brief presentation of the SIKE protocol. We refer readers to [17,16] for more detailed explanation of the supersignular isogeny problem and the base key-exchange protocol which the SIKE is constructed upon.

2.1 Isogenies of Elliptic Curves

Let p be a prime of the form $p = 2^{e_A}3^{e_3} - 1$, and let E be a supersingular elliptic curve defined over a field of characteristic p. E can be also defined over $\mathbb{F}_{p^2}$ up to its isomorphism. An isogeny $\phi : E \to E'$ is a non-constant map from E to E' which translates the identity into the identity. An isogeny map is defined by its degree and kernel. The degree of an isogeny is its degree as morphism. An isogeny with degree ℓ map is called ℓ-isogeny. Let G be a subgroup of points on E which contains $\ell + 1$ cyclic subgroups of order ℓ. This subgroup is the torsion group $E[\ell]$ and each element of this group is corresponding to an isogeny of degree ℓ; accordingly, an isogeny also can be identified by G, i.e., the kernel of isogeny, and it can be computed using Vélu's formula [32]. We denote this map as $\phi : E \to E'/\langle G \rangle$. Vélu's formula can only compute the isogeny of small degrees, while isogeny-based cryptography requires the evaluation of large-degree isogenies on curves. An efficient recursive strategy of computing large-degree isogeny is described in [17], by representing full binary trees and dynamic algorithms. The proposed strategy is adopted inside the SIKE software to compute large-degree isogeny.

Isogenies of elliptic curves divide isomorphic curves into isomorphism classes over $\mathbb{F}_{p^2}$. These classes of curves are categorized with their j-invariants [30]. This value is unique for isomorphic curves of the same class.

SIKE implementation is constructed on Montgomery curves, taking advantage of their fast and compact arithmetic. Let E be a Montgomery curve which defined by $E : By^2 = x^3 + Ax^2 + x$ equation. The j-invariant of E can be derived by the following equation:

$$j(E) = \frac{256(A^2 - 3)^3}{A^2 - 4}. \tag{1}$$

Therefore, in order to verify whether two elliptic curves are on the same class of isomorphisms, we can evaluate their j-invariant values. This feature is exploited to construct the supersingular isogeny key encapsulation mechanism [16], where two parties compute two isomorphic curves of the same class, and the shared secret is computed as the shared j-invariant values. Moreover, from (1), in order to compute the j-invariant of a Montgomery curve E, we only need to push the curve coefficient A into the formula. Further, we can compute the curve coefficient using the x-abscissas of two points x_P and x_Q on the curve as follows:

$$A = \frac{(1 - x_P x_Q - x_P x_R - x_Q x_R)^2}{4 x_P x_Q x_R} - x_P - x_Q - x_R, \tag{2}$$

where $R = P - Q$ is also a point on E. Therefore, the j-invariant of a Montgomery curve can be evaluated using the x-abscissas of two points and their difference. The above abstraction is used inside the supersingular isogeny encryption procedure which is explained in the next section.

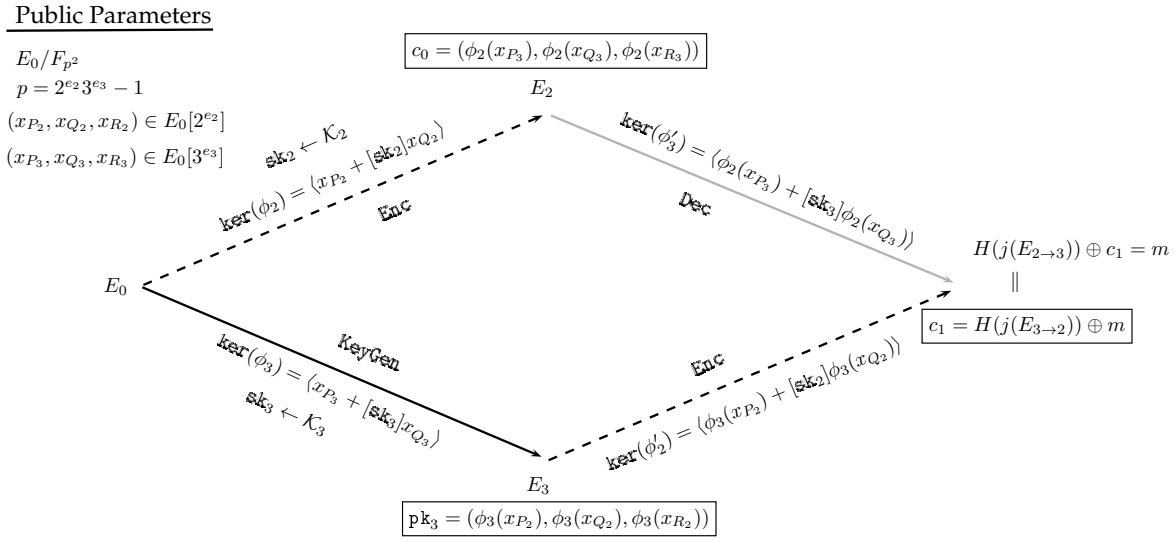

Fig. 1: Supersingular isogeny PKE protocol.

2.2 Supersingular Isogeny Key Encapsulation (SIKE) Mechanism

Public Parameters SIKE protocol [16], like any other supersingular isogeny-based scheme, is defined over a set of public parameters. These parameters need to be agreed by the parties prior to the key encapsulation mechanism and they are listed as follows:

- A prime p of the form $p = 2^{e_2}3^{e_3} - 1$, where e_A, e_B are two positive integers. The corresponding finite field is defined over $\mathbb{F}_{p^2}$. This explicit form is required for two main reasons. First, the isogeny computations using Vélu's formula need to be constructed over two different torsion subgroups, i.e., $E[2^{e_2}]$ and $E[3^{e_3}]$ of points on a starting curve for each party. Second, for efficiency reasons, primes of this form are Montgomery-friendly primes and they provide faster arithmetic [9].
- A starting supersingular Montgomery curve E_0 defined over $\mathbb{F}_{p^2}$.
- Two sets of generators, i.e., 3-tuple x-coordinates from $E_0[2^{e_2}]$ and $E_0[3^{e_3}]$. Note that, as it is discussed in detail in [9,16], the entire protocol can be defined and implemented only using two x-coordinates bases of each torsion subgroup, i.e., $\{x_{P_2}, x_{Q_2}\} \in E[2^{e_2}]$ and $\{x_{P_3}, x_{Q_3}\} \in E[3^{e_2}]$; however, for efficiency reasons, two auxiliary x-coordinates are used to encode these bases, i.e., $x_{R_2} = x_{P_2} - x_{Q_2}$ and $x_{R_3} = x_{P_3} - x_{Q_3}$.

In order to describe the SIKE protocol, first, we need to understand the supersingular isogeny public-key encryption scheme [17], because key encapsulation and decapsulation algorithms are defined based on supersingular isogeny encryption and decryption operations.

Supersingular Isogeny Public-key Encryption In this section, we recall the public-key encryption method from the isogenies of supersingular elliptic curves which was first introduced in [17], and it is recently described with some changes for achieving better performance in [16].

Similar to any public-key encryption schemes, supersingular isogeny PKE contains three main operations, i.e., key-generation (`KeyGen`), encryption (`Enc`), and decryption (`Dec`). Fig. 1 illustrates these operations from the isogeny map prespective.

Key Generation. The secret-key `sk`, is a random positive integer which is randomly generated from a key-space corresponding to each torsion subgroup's order. We denote them as $\mathcal{K}_2 = \{0, ..., 2^{e_2} - 1\}$ and $\mathcal{K}_3 = \{0, ..., 3^{e_3} - 1\}$, accordingly. Next, the secret key and the x-coordinates of generators construct the kernel of the first isogeny ϕ_ℓ such that $x_{S_\ell} = \langle x_{P_\ell + [\mathtt{sk}_\ell]Q_\ell} \rangle$, where $\ell \in \{2, 3\}$ is the degree of isogeny. Subsequently, public-key `pk` is computed by evaluating the isogeny of (2^{e_2} or 3^{e_3}) iteratively from the small isogeny evaluation and point multiplication as it is discussed previously. Eventually, public-key contains a 3-tuple of x-coordinates, i.e., $\mathtt{pk}_m = (x_{P_m}, x_{Q_m}, x_{R_m})$, where $x_{R_m} = x_{P_m} - x_{Q_m}$ and m is the degree of the isogeny. In Fig. 1 we choose `sk` from $\mathcal{K}_3$; therefore, the key generation algorithm maps E_0 to E_3.

Encryption. The encryption procedure encrypts an n-bit message m from a message space $\mathcal{M} = \{0,1\}^n$, and generates two ciphertexts c_0 and c_1. As it will be discussed later, in the case of key encapsulation mechanism, we need to generate a random string along with the encryption process for

security reason. The secret-key which is generated in the key generation procedure is randomly generated from either $\mathcal{K}_2$ or $\mathcal{K}_3$. Let $\mathtt{sk}_3$ be the secret-key from key generation algorithm from $\mathcal{K}_3$, and $\mathtt{pk}_3$ be its corresponding public key. We choose the other keyspace to generate the randomness inside the encryption algorithm, i.e., $\mathcal{K}_2$. The first ciphertext c_0 is generated as a 3-tuple of x-coordinates from evaluating the large degree 2^{e_2}-isogeny, starting from E_0 using the kernel ϕ_2 which is computed as $x_{S_2} = \langle x_{P_2+[\mathtt{sk}_2]Q_2} \rangle$.
The second ciphertext is generated after 3 steps:

1. First, the j-invariant of curve $E_{3\to 2}$ is evaluated by computing the large-degree 2_2^e-isogeny, initializing from the kernel $\phi_2' = \langle \phi_3(x_{P_2}) + [\mathtt{sk}_2]\phi_3(x_{Q_2}) \rangle$, and a starting curve E_3 which is retrieved from $\mathtt{pk}_3$ using equation (2).
2. Second, the hash of the retrieved j-invariant value from Step 1 is computed.
3. Computing the ciphertext as $c_1 = H(j) \oplus m$.

The computed pair of ciphertexts is the output of encryption procedure.

Decryption. Decryption algorithm computes m from (c_0, c_1) using $\mathtt{sk}_3$. First, we compute the j-invariant of curve $E_{2\to 3}$ which is computed by evaluating 3_3^e-isogeny of a starting curve E_2 and the kernel $\phi_3' = \langle \phi_2(x_{P_3}) + [\mathtt{sk}_3]\phi_2(x_{Q_3}) \rangle$. Note that E_2 is the evaluated curve from 3-tuple x-coordinate c_0. Second, we compute hash of the computed j-invariant using the same hash function used in encryption algorithm. Finally, we retrieve the message using $m = H(j) \oplus c_1$.

Key Encapsulation Mechanism A key encapsulation mechanism contains 3 main functions: key-generation, encapsulation, and decapsulation. In this section, we describe the SIKE protocol briefly based on the supersingular isogeny public-key encryption algorithm. We refer readers to [14] for more details.

Key Generation. Similar to `PKE` protocol, the key generation algorithm generates a secret-key from keyspace $\mathcal{K}_3$ and computes the corresponding 3-tuple x-coordinates $\mathtt{pk}_3$ by evaluating 3^{e_2}-degree isogeny from starting curve E_0. Moreover, an n-bit secret random message $s \in \{0,1\}^n$ is generated and concatenated to $\mathtt{sk}_3$ and $\mathtt{pk}_3$ to construct the SIKE secret-key $\mathtt{sk}_3$. The generated $\mathtt{pk}_3$ and $\mathtt{sk}_3$ are the output of this operation.

$$E_0 \to E_3/\langle x_{P_3} + [\mathtt{sk}_3]x_{Q_3} \rangle \to \mathtt{sk}_3 : (s, \mathtt{sk}_3, \mathtt{pk}_3). \tag{3}$$

Key Encapsulation. Key encapsulation defines on top of the supersingular isogeny encryption method. The input of this operation is the public-key $\mathtt{pk}_3$ which is generated in key-generation procedure. First, an n-bit random string $m \in \{0,1\}^n$ is generated and concatenated with the public-key $\mathtt{pk}_3$. Next, the result is hashed using a custom-`SHAKE256` (`cSHAKE256`) hash function G. This hash value is the *ephermeral secret-key* r, and it is pushed along with $\mathtt{pk}_3$ into encryption function to construct the SIKE ciphertext. The hash function H inside the encryptor is also a `cSHAKE256` function. The generated ciphertexts are further concatinated with m and hashed to generate the secret shared-key K:

$$\begin{gathered} \mathtt{Enc}(\mathtt{pk}_3, m, G(m \parallel \mathtt{pk}_3)) \to (c_0, c_1) \\ H(m \parallel (c_0, c_1)) \to K. \end{gathered} \tag{4}$$

Key Decapsulation. The decapsulation algorithm computes the shared-key K from the outputs of equations (3) and (4). First, 2-tuple ciphertext is decrypted using secret-key $\mathtt{sk}_3$ and hashed to retrieve m'. Further, m' is concatenated with public-key $\mathtt{pk}_3$ and hashed using the G function to retrieve ephemeral secret-key r'.

$$\begin{gathered} \mathtt{Dec}(\mathtt{sk}_3, (c_0, c_1)) \to m' \\ G(m' \parallel \mathtt{pk}_3) \to r'. \end{gathered}$$

Next, c_0' is computed by evaluating 2_2^e-isogeny of starting curve E_0 using the kernel $\langle x_{P_2} + [r']x_{Q_2} \rangle$:

$$E_0 \to E_2/\langle x_{P_2} + [r']x_{Q_2} \rangle \to c_0'.$$

The final correction is performed by comparing the c_0 value with c_0' and if they are equal, the shared-key K is computed as $K = H(m' \parallel (c_0, c_1))$, otherwise $K = H(s \parallel (c_0, c_1))$.

Table 1: SIKE finite field parameters [16]

$p = 2^{e_2}3^{e_3} - 1$	Length (bits)	$\min(\sqrt[3]{2^{e_2}}, \sqrt[3]{3^{e_3}})$	Quantum Security
$2^{250}3^{159} - 1$	503	1.26×2^{83}	83
$2^{372}3^{239} - 1$	751	1.00×2^{124}	124
$2^{486}3^{301} - 1$	964	1.02×2^{159}	159

Table 2: SIKE base field operation counts over different parameter sets

Scheme	mult.	red.	add.	sub.
`SIKEp503`	195,889	149,138	56,978	83,142
`SIKEp751`	307,946	234,253	88,764	131,618
`SIKEp964`	408,786	310,707	117,666	172,910

2.3 SIKE Implementation Parameters

The reference implementation of SIKE [16] contains three sets of parameters which correspond to three finite fields, denoted as $\mathbb{F}_{p503}$, $\mathbb{F}_{p751}$, and $\mathbb{F}_{p964}$ because of their bit-length. Our optimized software is forked from the reference implementation and it offers the same security levels using the same parameters. Table 1 presents the information regarding each prime and its provided quantum security level. The starting elliptic curve of the reference implementation is defined as $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$, with the cardinality equal to $\#E_0 = (2^{e_2}3^{e_3})^2$ and $j(E_0) = 1728$. This special instance of the Montgomery curve is chosen for efficiency reason, while any other supersingular curve can be used inside the protocol. We remark that the best known quantum attack on isogeny-based cryptography is based on claw-finding algorithm using quantum walks [31], which theoretically can find the isogeny between two curves in $O(\sqrt[3]{\ell^{e_\ell}})$, where ℓ^{e_ℓ} is the size of the isogeny kernel; accordingly, the provided quantum security level for each prime in SIKE is determined by the minimum bit-length of each isogeny kernel, i.e., $\min(\sqrt[3]{2^{e_2}}, \sqrt[3]{3^{e_3}})$. Detailed discussion about the security of the SIKE protocol is out of scope of this work and we refer the readers to [16] for further details.

3 Optimized Implementation on ARMv8

In this section, we explain our design approach and techniques. Our state-of-the-art implementation technique concentrates on the field multiplication, since it is the most expensive operation inside the SIKE protocol.

3.1 SIKE Performance Profiling

The main arithmetic operations inside the SIKE protocol are elliptic curve point arithmetic over quadratic extension field $\mathbb{F}_{p^2}$. Since the bit-length of the primes are smaller than the multiple of word size, in the reference implementation, optimization techniques such as lazy reduction are exploited to improve the overall performance of $\mathbb{F}_{p^2}$ arithmetic. As a result, finite field multiplication and reduction are implemented separately, and the reduction is delayed as much as possible to minimize the number of field arithmetic. In order to optimize the SIKE reference implementation, first we need to profile the number of operation counts for underlying field arithmetic. Since the isogeny computations for different degrees require different number of operations in corresponding with the isogeny graph, the number of field operations is different for each security level. Table 2 presents the number of field arithmetic operations in the SIKE protocol over different finite fields. This helps us to concentrate on the performance-bottleneck operations and try to improve them further compared to previous optimized implementations. As mentioned above, using lazy reduction technique reduces the number of field reduction to field multiplication significantly. Moreover, projective coordinates arithmetic replaces all the field inversions with extra multiplications and the performance of multiplier directly affects the overall performance of the protocol. Therefore, in this work, we focus on the efficiency of the field multiplication and engineer a highly-optimized multiplier for each quantum-security level. We find different levels of Karatsuba multiplication very effective to improve the performance compared to operand-scanning method, while we can exploit a parallel strategy to compute the multiplication of each half using mixed A64 and ASIMD assembly instructions. We elaborate more on this technique in this paper , but first we describe the most relevant architectural capabilities of the target platform which are considered in the design of our optimized library.

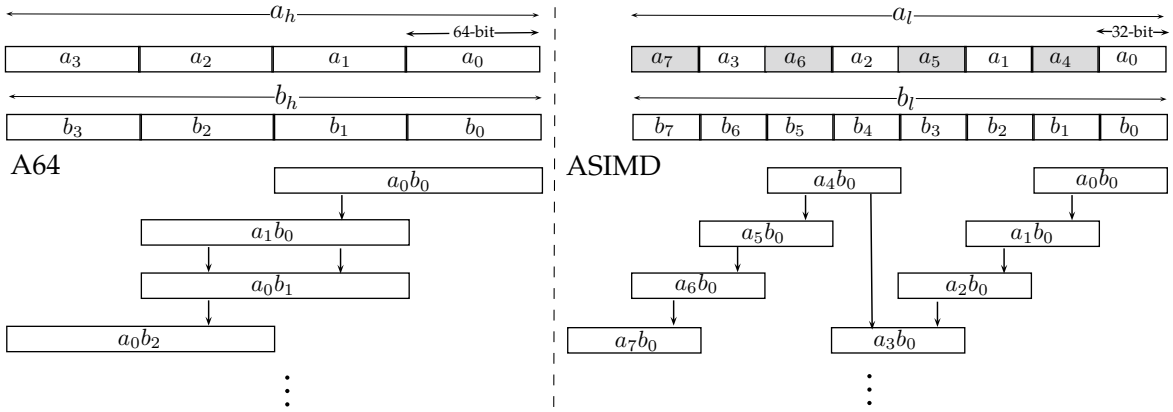

Fig. 2: 512-bit mixed-multilication implementation overview.

3.2 Target Architecture

We present a new design of arithmetic implementation on ARMv8 platforms by combining A64 and ASIMD instructions. To have these units working in parallel, the target platform should support *out-of-order* super-scalar pipeline. This feature is provided by ARMv8 high-performance cores such as Cortex-A57, Cortex-A72, and Cortex-A73. Therefore, we expect to get the best performance results of our *mixed* implementation on these family of processors, while our A64 implementation is anticipated to outperform the mixed version on power-efficient cores such as Cortex-A53.

ARMv8 Cortex-A57 and Cortex-A72 This family of processors are designed for the high-performance applications where they combine with power-efficient cores to achieve power-performance efficiency in the ARM `big.LITTLE` technology. Instructions are fetched and decoded in order into internal micro-operations (μops), and issued out-of-order to one of eight execution pipelines [4]; accordingly, two separate pipelines are dedicated for A64 and ASIMD μops. Once A64 instructions are dispatched through the integer pipeline, they are queued to be executed. Meanwhile, ASIMD pipeline can execute vector operations in the background when the A64 pipeline is stalled. This approach removes pipeline stalls to some extent and it is expected to boost the performance of arithmetic. We exploited this capability to implement optimized $\mathbb{F}_p$ multiplication.

We implement the field multiplication using two different strategies for each level of security. First, we implemented the straightforward A64 implementation using one-level Karatsuba technique. Next, we designed a mixed version of Karatsuba multiplication to be utilized in our target architecture.

3.3 Karatsuba Multiplication in A64

Previous implementations of field multiplication over $\mathbb{F}_{p751}$ [9,15] and $\mathbb{F}_{p964}$ [15] are based on Comba-based and two-level Karatsuba multiplication, respectively. In this work, we propose one-level Karatsuba multiplication method for both $\mathbb{F}_{p503}$ and $\mathbb{F}_{p751}$, while we follow two-level implementation over $\mathbb{F}_{p964}$ for achieving better performance as discussed in [15]. In particular, we achieved better performance results over $\mathbb{F}_{p751}$ using Karatsuba multiplication compared to previous Comba-method implementations.

Using A64 assembly instructions, one-level Karatsuba implementation is straightforward; field operands are represented in radix-2^{64} and two n-bit field elements, e.g., a and b are divided into two $\frac{n}{2}$-bit halves, i.e., $a_h 2^{\frac{n}{2}} + a_l$ and $b_h 2^{\frac{n}{2}} + b_l$. Consequently, one n-bit long multiplication is replaced with three $\frac{n}{2}$-bit multiplications, at the cost of extra additions/subtractions: $c = (a_h + a_l).(b_h + b_l) - a_h b_h - a_l b_l$.

Multiplication implementation in A64 starts with loading input operands into general 64-bit registers using `ldp` instruction which can load a pair of 64-bit data in each instruction. Next, the multiplication of 64-bit registers is implemented using `mul` and `umulh` instructions, computing the low and high halves of the 128-bit result, respectively. The intermediate additions, subtractions, and carry propagations are performed using `adcs` and `sbcs`. Eventually, the result is stored back to memory iteratively in pairs of 64-bit wide using `stp` instruction.

3.4 Mixed A64-ASIMD Karatsuba Multiplication

One of the benefits of using divide-and-conquer method such as Karatsuba multiplication is that the parts which build the final result, e.g., $(a_h+a_l).(b_h+b_l)$, $a_h b_h$, and $a_l b_l$ can be computed independently.

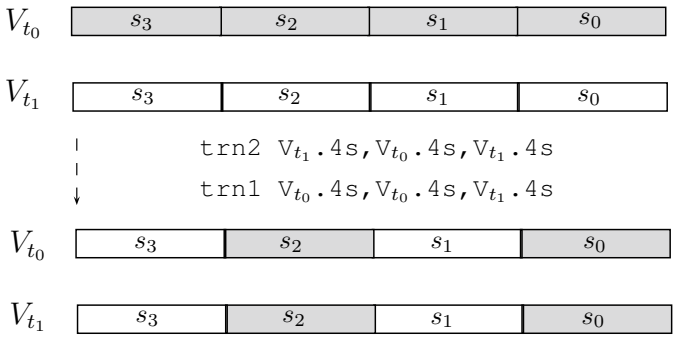

Fig. 3: Transposition of data into two vectors to handle carry-overflow.

This fact is in contrast with other multiplication methods such as Comba-method which requires the carry to be propagated step-by-step between each slot of the final result; as a result, the divided parts can be computed in parallel and combined at the end. In our mixed version of implementation, we exploit this advantage and compute a_hb_h and a_lb_l concurrently. We choose to implement a_hb_h using A64 and a_lb_l using ASIMD assembly instructions. Fig. 2 illustrates the overall overview of our mixed A64-ASIMD multiplication for 512-bit input operands.

As it is already discussed in details [15,14], it is rather inefficient to attempt ASIMD implementation for field arithmetic on 64-bit ARM due to the smaller radix representation of operands, i.e., radix-2^{32} and therefore extended number of arithmetic operations. However, in our engineered mixed implementation, many pipeline stalls are removed; this may result in improvement in overall performance the multiplication.

The first step to design our mixed multiplier is to choose what algorithm to use for each half of multiplication, i.e., a_hb_h and a_lb_l. For A64 implementation, we found the Comba-multiplication method fast and optimum. In particular, since the multiplication size is cut in half ($\frac{n}{2}$-bit), we have access to redundant number of 64-bit general registers that can be used to implement the operand-scanning method without any need to load/store data back and forth in the middle of process. On the other hand, for the ASIMD implementation of a_lb_l, Comba-multiplication is not a suitable option due to the lack of carry-propagation instruction. Previous works on the efficient implementation of multi-precision multiplication on ARMv7 NEON [27,20] present an efficient field multiplication using parallel school-book method. We adopted the same strategy; however, since the vector manipulation in ARMv8 ASIMD is totally different from ARMv7 NEON, we customized our version of school-book multiplication on ARMv8 platforms.

A64-based product-scanning method is implemented by loading the most significant halves of input operands, and computing each slot of the result by multiplication and addition of corresponding input words. ASIMD-based school-book multiplication, however, is more complicated to implement. We explain the implementation steps in detail in the following:

1. After loading the least-significant halves of input operands into vectors using `ld1` instruction, one of the input vectors is rearranged in correspond with the intermediate additions positions (highlighted words in Fig. 2) using transpose instructions `trn1` and `trn2`. Note that, ASIMD multiplication instructions compute two 32×32-bit multiplications concurrently, i.e., a_0b_0 and a_4b_0 are computed using one multiplication instruction in parallel.
2. Each 32-bit limb of "non-shuffled" operand is multiplied to the entire shuffled operand in each step using `umull` and `umull2` instructions. These instruction computes the multiplication of a 32-bit limb to the first and second halves of a 128-bit ASIMD vector, respectively. We use these instructions only in the first step of the multiplication process since we need no addition. For the next steps, we exploit `umlal` and `umlal2` instructions which compute the multiplication and addition at the same time.
3. Vectorized additions are performed in parallel and can add two 128-bit vector using only one `add` instruction. However, vectorized addition does not handle carry-propagation between vector slots; accordingly, intermediate multiplication results need to be transposed from 4×32-bit values into 2×32-bit ones, while the remaining two slots are preserved for carry overflows. Fig. 3, shows this process for a 128-bit wide ASIMD vector. This transposition is performed by compounding the multiplication result vector (V_{t_0}) with a zero vector (V_{t_1}) using `trn1` and `trn2` instructions.
4. The ASIMD-based multiplication performs Steps 2 and 3 iteratively for each 32-bit word of the second operand. In the final step we compute the last additions and carry propagations and move the result vectors into general registers using `umov` instruction to perform subtraction ($a_hb_h - a_lb_l$).

Table 3: Performance results (presented in millions of clock cycles) of the proposed softwares in comparison with reference implementation on ARMv8 platforms. (Benchmarks were obtained on 1.95 GHz Cortex-A57 and 2.4 GHz Cortex-A72 cores running Android 7.1.1 and 8.1.0, respectively)

Scheme	Operation	A64 assembly		(ASIMD + A64) assembly		Optimized C	
		Cortex-A57	Cortex-A72	Cortex-A57	Cortex-A72	Cortex-A57	Cortex-A72
SIKEp503	KeyGen.	31.2	31.2	31.2	33.6	150.1	147.5
	Encap.	50.7	50.4	52.6	55.2	245.7	245
	Decap.	54.6	54.1	56.5	57.6	261.3	260
	Total	**136.5**	**135.7**	140.3	146.4	657.1	652.5
SIKEp751	KeyGen.	101.4	100.8	99.4	98.4	528.4	512.5
	Encap.	163.8	162.6	161.8	161.2	858	835
	Decap.	175.5	174.8	173.5	172.4	922.3	892.5
	Total	440.7	438.2	**434.7**	**432**	2,308	2,255
SIKEp964	KeyGen.	222.3	220.8	216.4	214.2	1,201.2	1,167.6
	Encap.	374.4	372	362.7	361.1	2,020.2	1,918.8
	Decap.	393.9	393.6	382.2	380.7	2,131.3	2,090.9
	Total	990.6	986.4	**961.3**	**956**	5,352.7	5,177.3

We implement the rest of Karatsuba multiplication operations, i.e., $(a_h + a_l).(b_h + b_l)$ and subtractions, using A64 instruction, taking advantage of its 64-bit wide arithmetic. Furthermore, we design 512-, 768-, and 1024-bit mixed Karatsuba multiplication using the above technique and integrate them into the SIKE software, providing three different quantum security levels.

We remark that since all the curve arithmetic in SIKE implementation is performed on Montgomery space, the most optimized algorithm for reduction is Montgomery reduction which is already used in the SIKE implementation submission. Moreover, SIKE primes, i.e., $\mathbb{F}_{p503}$, $\mathbb{F}_{p751}$, and $\mathbb{F}_{p964}$, have a special form which is utilized to improve the reduction algorithm by eliminating several single-precision multiplications; we refer the reader to [9,15] for more details. Therefore, we believe the most efficient implementation of reduction algorithm on ARMv8 is based on product-scanning method using A64 assembly instructions.

4 Optimized SHA-3 implementation

SIKE reference implementation requires three hash functions F, G, and H which are used inside encapsulation and decapsulation algorithms. These hash functions are all instances of `cSHAKE256` with different custom input strings which is based on SHA-3 library and specified by NIST. Compared to isogeny computations, the computational cost of these functions is negligible and they barely affect the overall performance of the SIKE protocol. However, in this work, we focus on using the most efficient implementation of this protocol on ARMv8 platforms; therefore, we replace the generic implementation of SHA-3 with the hand-written assembly version developed by OpenSSL[6] project authors in our software. The essential function inside the SHA-3 is the `Keccak-1600` function which performs the core permutations. Note that, the optimized implementation of this function is developed using A64 assembly instructions without taking advantage of ASIMD vectors for the following reason. Addressing 64-bit lanes of ASIMD vectors is not as trivial as 32-bit NEON on ARMv7 processors. In particular, we found it is rather complicated to perform `rotate` operation using available ASIMD instructions on 64-bit lanes which adds up significant number of clock cycles to the overall timing results. Therefore, 64-bit ASIMD implementation of `Keccak-1600` is slower than A64 general register implementation on ARMv8 processors.

5 Implementation Details and Performance Results

In this section, we present our benchmark procedure and provide SIKE performance evaluation results on two famous ARMv8 family of processors, i.e., Cortex-A57 and Cortex-A72.

5.1 Implementation Details

As it is mentioned before, our optimized implementation targets high-performance ARMv8 processors which support out-of-order pipeline. We use a Huawei Nexus 6P cellphone, running Android

[6] Available in: `https://github.com/openssl/openssl`

Table 4: Performance comparison of this work with the previous ARMv8 optimized implementation of SIKEp751 on a Cortex-A57 core presented in millions of clock cycles

Implementation	KeyGen	Encap.	Decap.	Total
This Work	**99.4**	**161.8**	**173.5**	**434.7**
SIDHv3.0*	109.2	179.4	193.2	481.8

* Supports optimized ARMv8 implementation of SIKEp751

Nougat 7.1.1 and a Google Pixel 2, running Android Oreo v8.1.0, for benchmarking our software on Cortex-A57 and Cortex-A72, respectively. The binaries are cross-compiled with `clang 5.0.3` using `-O3 -fomit-frame-pointer -pie` flags and run via `adb-shell`. The `-pie` flag is used to generate position independent executables which can be run on cellphones. We benchmarked the executables on one high-performance core using `taskset` command to ensure power-efficient cores were not involved.

Our optimized software libraries using both A64 and mixed-assembly are publicly available[7] for different levels of security.

5.2 Results and Discussion

Table 3 presents benchmark results of our SIKE implementation on target platforms for different security levels. The results highlight the performance of our optimized ARMv8 assembly implementation using A64 assembly and mixed assembly in comparison with the portable optimized C reference implementation [16]. Results are averaged over 10^3 iterations and converted to clock cycle counts corresponding to the target processor frequency. Based on the benchmark results, the optimized arithmetic libraries provide roughly $5\times$ faster results than generic implementation on the target platforms. Moreover, we notice that for larger finite fields, our mixed ASIMD/A64 version outperforms the A64 optimized library by a very small factor, while in relatively smaller fields, the A64 implementation is more efficient. This highlights the fully pipeline utilization that we obtain using mixed assembly implementation. However, as it is discussed before, since ASIMD arithmetic operations are performed in radix-2^{32}, the overall performance improvement is not remarkable. We also observe that the clock cycle count on Cortex-A72 core is slightly less than Cortex-A57. This performance enhancement is related to architecture design improvements in Cortex-A72 family of processors which contain many micro-architecture updates for performance improvement compared to its prior generation.

In order to evaluate the efficiency of Karatsuba multiplication over $\mathbb{F}_{p751}$, we compared our mixed optimized library with the only available[8] highly-optimized ARMv8 SIKE implementation. We compared the performance of the SIDH v3.0 which supports optimized ARMv8 SIKE implementation with our software for the only available security level, i.e., `SIKEp751`. We have noticed the available ARMv8 implementation of `SIKEp503` inside the SIDHv3.0 library is implemented in C and therefore the performance comparison with this work is not fair. Table 4 presents the performance evaluation of the SIKE implementation inside SIDHv3.0 in comparison with this work. The presented benchmarks show roughly 10% performance improvement of our library over highly-optimized previous library for the same field size. We state that the performance improvement of this results corresponds to different multiplication algorithm and implementation technique that we use in this work. Moreover, it proves the fact that ARMv8 vector instructions can still be useful inside field arithmetic operations. In particular, using mixed implementation and taking advantage of 256-bit wide ASIMD vectors can result in performance improvement over larger fields. That is because the scarce number of available 64-bit general registers enforces an excessive number of memory-register load and stores instructions, and using vectors will reduce these costly operations, and improve the overall performance.

6 Conclusion

The main motivation behind this work was to push the performance bar of the post-quantum SIKE protocol further on 64-bit ARM-powered embedded devices; we presented an optimized implementation of SIKE protocol on 64-bit high-performance ARM processors. We investigated different design of implementing multi-precision multiplication on ARMv8 processors, with and without using ASIMD vectorization. We showed that incorporating ASIMD implementation, mixed with A64 design, over

[7] `https://github.com/amirjalali65/armv8-sike`

[8] `https://github.com/microsoft/pqcrypto-sidh` (accessed in Feb. 2018)

relatively large finite fields, can improve the overall performance of projective coordinates cryptography protocols such as SIKE.

Our proposed approach can be used inside any other cryptography primitives which require significant amount of field multiplication over large primes. Moreover, we benchmarked our implementations on two popular Android cellphones to evaluate the practical efficiency of the SIKE protocol. We compared our results with the previous SIKE optimized implementation to justify the performance benefits we gained using our proposed implementation techniques. We believe the proposed method can be used on other platforms which support out-of-order execution pipeline, improving the overall performance of the protocol by reducing pipeline stalls. We plan to investigate this possibility in the future works.

We hope this work motivates researchers and engineers to investigate further into the efficiency of the SIKE protocol on various platforms.

References

1. Ajtai, M.: Generating hard instances of lattice problems. In: Proc. ACM Symp. on Theory of computing. pp. 99–108. ACM (1996)
2. Ali, S., Cenk, M.: Faster residue multiplication modulo 521-bit mersenne prime and an application to ECC. IEEE Trans. on Circuits and Systems **65-I**(8), 2477–2490 (2018)
3. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange-a new hope. Tech. rep., Cryptology ePrint Archive, Report 2015/1092 (2015), `http://eprint.iacr.org` (accessed Feb. 2018)
4. ARM Limited: Cortex-A57 Software Optimization Guide (2016), `http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/cortex_a57_software_optimization_guide_external.pdf` (accessed Feb. 2018)
5. Azarderakhsh, R., Jao, D., Leonardi, C.: Post-quantum static-static key agreement using multiple protocol instances. In: Proc. Int. Conf. on Selected Areas in Cryptography. pp. 45–63. Springer (2017)
6. Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., Stebila, D.: Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In: Proc. ACM SIGSAC Conference on Computer and Communications Security. pp. 1006–1018. ACM (2016)
7. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In: Proc. IEEE Symp. on Security and Privacy. pp. 553–570. IEEE (2015)
8. Charles, D.X., Lauter, K.E., Goren, E.Z.: Cryptographic hash functions from expander graphs. J. Cryptology **22**(1), 93–113 (2009)
9. Costello, C., Longa, P., Naehrig, M.: Efficient algorithms for supersingular isogeny Diffie-Hellman. In: Advances in Cryptology. Springer (2016)
10. De Feo, L., Jao, D., Plût, J.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. J. of Mathematical Cryptology **8**(3), 209–247 (2014)
11. Faz-Hernández, A., López, J., Ochoa-Jiménez, E., Rodríguez-Henríquez, F.: A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. IEEE Trans. Comput. (2017)
12. Galbraith, S.D., Petit, C., Silva, J.: Signature schemes based on supersingular isogeny problems. Tech. rep., Cryptology ePrint Archive, Report 2016/1154 (2016 (accessed Feb 2018))
13. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Proc. Int. Symp. on Algorithmic Number Theory. pp. 267–288. Springer (1998)
14. Jalali, A., Azarderakhsh, R., Mozaffari-Kermani, M.: Efficient post-quantum undeniable signature on 64-bit arm. In: Proc. Int. Conf. on Selected Areas in Cryptography. pp. 281–298. Springer (2017)
15. Jalali, A., Azarderakhsh, R., Mozaffari Kermani, M., Jao, D.: Supersingular isogeny diffie-hellman key exchange on 64-bit arm. IEEE Trans. Depend. Secure Comput. (2017)
16. Jao, D., Azarderakhsh, R., Campagna, M., Costello, C., Jalali, A., Koziel, B., LaMacchia, B., Longa, P., Naehrig, M., Renes, J., Soukharev, V., Urbanik, D.: Supersingular isogeny key encapsulation. NIST submissions (2017)
17. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Proc. Int. Workshop on Post-Quantum Cryptography. pp. 19–34. Springer (2011)
18. Jao, D., Soukharev, V.: Isogeny-based quantum-resistant undeniable signatures. In: Proc. Int. Workshop on Post-Quantum Cryptography. pp. 160–179. Springer (2014)
19. Koziel, B., Azarderakhsh, R., Kermani, M.M., Jao, D.: Post-quantum cryptography on FPGA based on isogenies on elliptic curves. IEEE Trans. on Circuits and Systems **64-I**(1), 86–99 (2017)
20. Koziel, B., Jalali, A., Azarderakhsh, R., Jao, D., Mozaffari-Kermani, M.: NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In: Proc. Int. Conf. on Cryptology and Network Security. pp. 88–103. Springer (2016)
21. Lange, T.: PQCRYPTO Project in the EU. NIST Workshop on Cybersecurity in a Post-Quantum World (2015), `http://csrc.nist.gov/groups/ST/post-quantum-2015/presentations/session7-lange-tanja.pdf` (accessed Feb. 2018)
22. Longa, P.: Fourqneon: Faster elliptic curve scalar multiplications on ARM processors. In: Selected Areas in Cryptography - SAC 2016 - 23rd International Conference,. pp. 501–519 (2016)

23. Matsumoto, T., Imai, H.: Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. In: Proc. Workshop on the Theory and Application of of Cryptographic Techniques. pp. 419–453. Springer (1988)
24. McEliece, R.: A public-key cryptosystem based on algebraic. Coding Thv **4244**, 114–116 (1978)
25. Merkle, R.C., Charles, R., et al.: Secrecy, authentication, and public key systems. PhD thesis, Stanford University (1979)
26. Rostovtsev, A., Stolbunov, A.: Public-key cryptosystem based on isogenies. IACR Cryptology ePrint Archive **2006**, 145 (2006)
27. Seo, H., Liu, Z., Großschädl, J., Choi, J., Kim, H.: Montgomery Modular Multiplication on ARM-NEON Revisited. In: Proc. Int. Conf. on Information Security and Cryptology. pp. 328–342. Springer (2014)
28. Seo, H., Liu, Z., Longa, P., Hu, Z.: SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 1–20 (2018)
29. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proc. Foundations of Comput. Science Symp. pp. 124–134. IEEE (1994)
30. Silverman, J.H.: The arithmetic of elliptic curves, vol. 106. Springer Science & Business Media (2009)
31. Tani, S.: Claw finding algorithms using quantum walk. Theoretical Computer Science **410**(50), 5285–5297 (2009)
32. Vélu, J.: Isogénies entre courbes elliptiques. CR Acad. Sci. Paris Sér. AB **273**, A238–A241 (1971)
33. Yoo, Y., Azarderakhsh, R., Jalali, A., Jao, D., Soukharev, V.: A post-quantum digital signature scheme based on supersingular isogenies. In: Proc. Int. Conf. on Financial Cryptography and Data Security. pp. 163–181. Springer (2017)