This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Supersingular Isogeny Diffie-Hellman Key Exchange on 64-bit ARM

Amir Jalali, Reza Azarderakhsh, *Member, IEEE*, Mehran Mozaffari Kermani, *Senior Member, IEEE*, and David Jao

Abstract—We present an efficient implementation of the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol on 64-bit ARMv8 processors for 125- and 160-bit post-quantum security levels. We analyze the use of both affine and projective SIDH formulas and provide a comprehensive analysis of both approaches based on the inversion-to-multiplication ratio. Implementation results show that regardless of security concerns, affine SIDH is competitive with the projective coordinates implementation, and even outperforms projective implementation in the final round of SIDH; however, projective SIDH shows better overall performance for the whole key exchange protocol. Notably, over larger finite fields, using optimized field multiplication leads to the much better performance of projective compared to affine formulas. We integrate our optimized software into the open quantum-safe OpenSSL library and compare our software with other available post-quantum primitives. The benchmark results on ARMv8 demonstrate speedup of up to 5X over the generic version of SIDH implementation which is available inside the OQS library for the same quantum security level. We observe that our highly-optimized implementation still suffers from a large number of operations for computing isogenies of elliptic curves. However, in terms of communication overhead, supersingular isogeny-based cryptosystem provides significantly smaller key size compared to its counterparts.

Index Terms—ARM assembly, elliptic curve cryptography, finite field, isogeny-based cryptosystems, OpenSSL, post-quantum cryptography.

1 Introduction

The possible impending arrival of large-scale quantum computers capable of practically performing Shor's algorithm [1] in the near future has motivated intensive research on the topic of post-quantum cryptosystems. NIST's recently-published draft report on post-quantum cryptography (PQC) [2] provides the guidance for researchers to develop practical candidates for post-quantum cryptosystems in various applications. PQC research deals with investigation and study of cryptographic algorithms that are believed to be secure against quantum attacks. There exist several promising quantum-resistant cryptography primitives which are claimed to be secure against quantum computer attacks. Among all of them, the NIST report identifies five classes of cryptographic primitives which are regarded as leading candidates for post-quantum cryptography, namely code-based cryptography [3], multivariate cryptography [4], hash-based cryptography [5], lattice-based cryptography including the NTRU encryption scheme [6], and isogeny-based cryptography [7].

In this paper, we consider the supersingular isogeny Diffie-Hellman (SIDH) key-exchange protocol which was first introduced by Jao and De Feo [8]. SIDH key exchange, like the classical Diffie-Hellman key exchange protocol, is based on the difficulty of solving a certain number-theoretic problem, in this case, to construct an isogeny of a particular degree between two given isogenous supersingular elliptic curves, defined over a finite field of characteristic p. To date, classical [9] and quantum [10] attacks on isogeny-based cryptosystems and its related problem, using claw-finding algorithms ([11] and [12]) solve this problem in

A. Jalali and R. Azarderakhsh are with the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL, USA. E-mail: {ajalali2016, razarderakhsh}@fau.edu.
M. Mozaffari Kermani is with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620, USA. E-mail: mmozaff@gmail.com.
D. jao is with the Mathematics Faculty at the University of Waterloo, Canada. E-mail: djao@uwaterloo.ca.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

$O(p^{1/4})$ and $O(p^{1/6})$ complexity on a classical and quantum computer, respectively. We believe SIDH is worth considering as an alternative to the four leading PQC candidates because it features significantly smaller key size [13], easier parameter generation, and a direct relationship between parameter size and security level. Additionally, it provides security assumptions based on elliptic curves and the isogeny construction problem which have already been well-studied in other contexts prior to the publication of SIDH such as cryptographic hash functions [14].

The SIDH key exchange protocol was first implemented by De Feo et al. [15]. Recently, a full-fledged, optimized implementation of SIDH has been proposed by Costello et al. [16] which is targeted Intel `x86-64` platforms, using projective coordinates formulae. Their implementation is a constant-time, almost inversion-free version of the SIDH protocol which yields efficient performance on Intel-based devices. However, on ARM platforms, the status of SIDH implementation is currently less well-defined. Early efforts by Azarderakhsh et al. [17] suffered from the excessive number of operations caused by adopting generic approaches for finite field arithmetic. Recently, a highly-optimized affine SIDH implementation on 32-bit ARMv7 using NEON vectorization by Koziel et al. [18] still showed an order of magnitude difference between Intel and ARM processors in terms of SIDH performance as measured by cycle counts (with the caveat that direct comparisons between different architectures with different word sizes are only marginally useful). Therefore, in this work, we address this gap and develop an optimized implementation of both projective and affine SIDH protocol for two quantum security levels on the high-performance 64-bit ARMv8 Cortex-A57 core; a massively popular platform with 65% worldwide market share among smart phones and growing importance in the server market [19]. We plan to make all our source codes publicly available in the near future.

To the best of our knowledge, this paper is the first efficient implementation of SIDH on ARMv8 processors over two different quantum security levels, and the first comparative implementation of SIDH using projective and affine curve arithmetic on any platform. Our main contributions are as follows:

- Efficient, hand-optimized assembly implementation of finite field arithmetic on ARMv8 devices, including comparison between the capabilities of ARMv8 Advanced Single Instruction Multiple Data (Adv. SIMD) and A64 instruction sets.
- Optimized algorithms and libraries for both projective and affine SIDH protocol on cutting-edge ARMv8 platforms.
- Efficient implementation of projective and affine SIDH for two different quantum security levels using implementation-friendly primes.
- Comprehensive analysis and comparison between affine and projective SIDH formulas in terms of performance and security.
- Detailed performance comparison of SIDH key exchange protocol with other post-quantum algorithms using the open quantum safe standard framework on ARMv8 processors.

2 Preliminaries

This section describes the abstract concepts and features of Diffie-Hellman key exchange from supersingular elliptic curve isogenies. We refer the readers to [8] for the detailed explanation of the protocol and underlying mathematics.

2.1 Supersingular Elliptic Curve Isogenies

Isogeny. Suppose E_1 and E_2 are elliptic curves defined over a finite field $\mathbb{F}_q$. An isogeny $\phi\colon E_1 \to E_2$ is a non-constant rational map defined over $\mathbb{F}_q$ such that ϕ is a group homomorphism from $E_1(\mathbb{F}_q)$ to $E_2(\mathbb{F}_q)$ [20]. Two elliptic curves E_1 and E_2 defined over $\mathbb{F}_q$ are isogenous if there exists an isogeny $\phi : E_1 \to E_2$ over $\mathbb{F}_q$; two elliptic curves are isogenous over $\mathbb{F}_q$ if and only if they have the same cardinality [21]. Isogenous elliptic curves are either all supersingular or all ordinary and in the case of SIDH protocol, only supersingular elliptic curves are considered. Moreover, among all supersingular elliptic curves, only curves with smooth orders are used in the protocol, because isogenies of exponentially large degree can be efficiently constructed based on compositions of low degree isogenies. More precisely, let p be a prime of the form $p = \ell_A^{e_A}\ell_B^{e_B} f \pm 1$ such that ℓ_A and ℓ_B are small prime numbers and f is an integer cofactor. In this case, a supersingular elliptic curve E can be efficiently constructed over $\mathbb{F}_{p^2}$, having smooth order $\left(\ell_A^{e_A}\ell_B^{e_B} f\right)^2$ [22].

ℓ-torsion subgroup. The ℓ-torsion subgroup of an elliptic curve ($E[\ell]$) is defined as the set of all geometric points P on the curve E, i.e., points defined

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

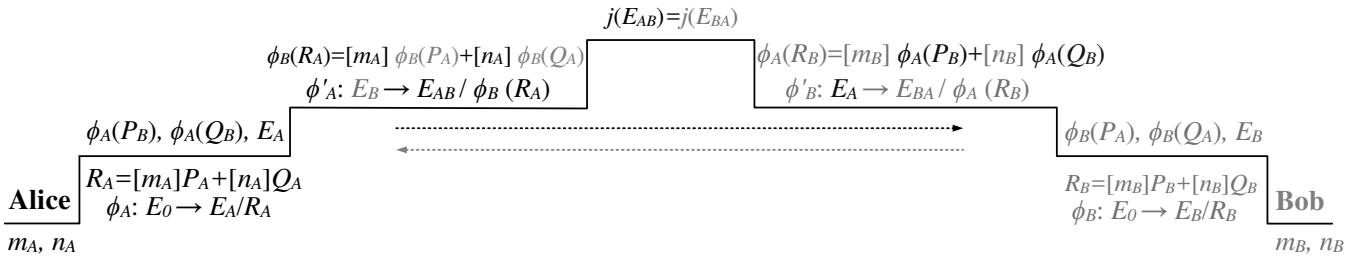

Figure 1: SIDH key exchange protocol [8]. Alice and Bob generate their public keys based on the public parameters. They exchange their public keys and both compute the final isomorphic curves. The computed curves have the same common j-invariant value.

over the algebraic closure of the field of definition of E, such that $\ell P = \mathcal{O}$. In case of supersingular curves and a prime of the form $p = \ell_A^{e_A}\ell_B^{e_B} f \pm 1$, the $\ell_A^{e_A}$-torsion and $\ell_B^{e_B}$-torsion groups are defined over $\mathbb{F}_{p^2}$ ($E[\ell_A^{e_A}], E[\ell_B^{e_B}] \subseteq E(\mathbb{F}_{p^2})$). Since $\ell_A, \ell_B \nmid p$, we can conclude $E\left[\ell_A^{e_A}\right] \cong (\mathbb{Z}/\ell_A^{e_A}\mathbb{Z}) \times (\mathbb{Z}/\ell_A^{e_A}\mathbb{Z})$ and $E\left[\ell_B^{e_B}\right] \cong (\mathbb{Z}/\ell_B^{e_B}\mathbb{Z}) \times (\mathbb{Z}/\ell_B^{e_B}\mathbb{Z})$ [20]. Now, let m and n be two random integers, and P, Q be two points on the supersingular elliptic curve E which generate $E\left[\ell_A^{e_A}\right]$ (or $E\left[\ell_B^{e_B}\right]$) as an Abelian group, of order $\ell_A^{e_A}$ (or $\ell_B^{e_B}$, respectively). A point of the form $[m]P+[n]Q$ has order dividing $\ell_A^{e_A}$ (or $\ell_B^{e_B}$) and this point generates a finite subgroup which we use as the kernel of an isogeny in the SIDH protocol. For a given finite subgroup R of E, an isogeny over E having kernel R, i.e., $\phi\colon E \rightarrow E/\langle R\rangle$, can be efficiently computed using Vélu's formulas [23].

2.2 Diffie-Hellman Key Exchange Protocol

In the first round of SIDH key exchange, Alice and Bob compute the graphs of isogenies of degree $\ell_A^{e_A}$ and $\ell_B^{e_B}$ separately to construct two isogenous elliptic curves [8]. In the second round, they "trade" kernels and each computes a second isogeny which lands on the same (isomorphic) curve, with the same j-invariant value, which can then be used as the shared secret key for a secure session. Alice and Bob decide on some public parameters before the key exchange procedure. These parameters are a supersingular elliptic curve E_0 defined over $\mathbb{F}_{p^2}$ with smooth cardinality $\left(\ell_A^{e_A}\ell_B^{e_B} f\right)^2$, two independent points P_A and Q_A which together generate the $\ell_A^{e_A}$-torsion subgroup, i.e., $E\left[\ell_A^{e_A}\right]$, and two independent points P_B and Q_B which together generate the $\ell_B^{e_B}$-torsion subgroup, i.e., $E\left[\ell_B^{e_B}\right]$. The key exchange protocol consists of four steps to generate the secure shared key between two parties as follows:

1) Alice chooses two secret integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, not both divisible by ℓ_A. She computes the double point multiplication $R_A = [m_A]P_A + [n_A]Q_A$ to compute a point R_A of order dividing $\ell_A^{e_A}$. Using her private key, i.e., (m_A, n_A), she computes the secret isogeny $\phi_A : E_0 \rightarrow E_A/\langle R_A\rangle$. She also computes the image points $\phi_A(P_B)$ and $\phi_A(Q_B)$. Her public key is the curve E_A, $\phi_A(P_B)$, and $\phi_A(Q_B)$ together which are sent to the other party.
2) Bob similarly chooses two secret integers $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$, not both divisible by ℓ_B. He computes the secret isogeny $\phi_B : E_0 \rightarrow E_B/\langle R_B\rangle$, using the kernel $R_B = [m_B]P_B + [n_B]Q_B$ of order dividing $\ell_B^{e_B}$, while (m_B, n_B) is his private key. He then computes $\phi_B(P_A)$ and $\phi_B(Q_A)$, and publishes his public key E_B, together with $\phi_B(P_A)$ and $\phi_B(Q_A)$.
3) Alice computes another isogeny $\phi'_A : E_B \rightarrow E_{AB}/\langle\phi_B(R_A)\rangle$, such that $\phi_B(R_A) = [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A)$. Bob computes the isogeny $\phi'_B : E_A \rightarrow E_{BA}/\langle\phi_A(R_B)\rangle$, whose kernel is the point $\phi_A(R_B) = [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B)$.
4) Alice and Bob now compute two isomorphic curves with the same j-invariants, i.e., $j(E_{AB}) = j(E_{BA})$, and they use this value as their shared secret key.

Figure 1 illustrates the required "steps" which Alice and Bob take to construct two isomorphic curves with the same j-invariants. A detailed discussion of SIDH security is given in [8], [15].

3 Implementation-friendly primes

The SIDH key-exchange protocol is constructed on the isogeny classes of supersingular elliptic curves with smooth orders, taking advantage of their special shape to compute isogenies of large degree efficiently. Since all the arithmetic are performed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

in Montgomery space, for efficiency reasons which will be discussed later, we choose to use primes of the form $p = 2^{e_A}\ell_B^{e_B} f \pm 1$ in our implementation. Recently, Bos et al. [24] provided a systematic overview of techniques to compute efficient arithmetic over SIDH key-exchange protocol. Their findings illustrate that for the fast arithmetic computation inside the SIDH key-exchange protocol, regardless of affine or projective formulas, it is more convenient to use the primes of the form $p = 2^{e_A}\ell_B^{e_B} \pm 1$, since they make the Montgomery reduction computation much more optimized because of their special shape. We set the second base prime of our smooth prime equal to $\ell_B = 3$ for the sake of simplicity in the implementation of our isogeny computations compared to larger degrees like $\ell_B = 19$ which is suggested in [24]. We also ensure that our implementation primes satisfy the security balance between Alice and Bob. That is, the difference between the size of the two prime powers is not too large, i.e., $|2^{e_A} - 3^{e_B}| < 2^{40}$.

Below, we discuss the properties of two moduli which we use in our implementation. The 751-bit prime was first proposed by Costello et al. [16] in the first version of projective SIDH implementation. We optimize their implementation on ARMv8 processors using our efficient arithmetic library to investigate the efficiency of their software on ARM processor and provide a full comparison of recent attempts on SIDH key-exchange implementations.

3.1 The Modulus $p751 = 2^{372} \cdot 3^{239} - 1$

The prime field provides 125-bit post-quantum security level and the arithmetic computations can be implemented efficiently using 12×64-bit words on 64-bit platforms. The first performance evaluation of projective SIDH key-exchange software using this prime was optimized for Intel processors. In this work, we independently implemented the same software using our ARMv8 optimized library to explore the performance of projective SIDH on 64-bit ARM processors as a reference. We remark that recently published optimized library [25] of projective SIDH on ARMv8 platforms over this finite field, and efficient ARMv8 filed arithmetic library [26] for SIDH key compression demonstrate comparable performance results with this work.

3.2 The Modulus $p964 = 2^{486} \cdot 3^{301} - 1$

We introduce our 964-bit implementation-friendly prime which provides theoretical 160-bit post-quantum security level. At first sight, this prime does not seem to be an efficient prime for implementing on 64-bit platforms since in the normal representation, it takes $16\times$64-bit words with only 4 bits in the last word. Nonetheless, as we will show in the following section, arithmetic over this prime can be efficiently implemented using redundant radix representation technique and delayed carry propagation. Furthermore, since the 7 least significant words of the prime are all equal to "1", as it will be discussed later, the Montgomery reduction implementation can be significantly optimized.

4 Implementation Methodology on ARMv8

This section presents the implementation methods and algorithms used in our finite field library. The proposed implementation methodology provides a detailed performance comparison between two different sets of instruction based on ARMv8 platform capabilities: A64 instructions using general registers and Adv. SIMD instructions using vectorization.

4.1 Targeted Architecture

The proposed implementation is optimized for the 64-bit Cortex-A series with ARMv8 support, with a special focus on the high-performance Cortex-A57 processor. This processor is equipped with fully *out-of-order* execution pipeline on both ARM and Adv. SIMD units. In many platforms, Cortex-A57 cores are combined with Cortex-A53 cores in the ARM big.LITTLE architecture, with the power-efficient A53 core used for standby tasks.

ARMv8 processors are capable of performing fast integer arithmetic using Adv. SIMD and A64 instructions. Table 1 presents a list of both Adv. SIMD and A64 arithmetic instructions that are required for finite field arithmetic implementation together with their latencies in clock cycles. In this table, execution latency is defined as the minimum latency seen by an operation, while execution throughput is referred to the maximum throughput of a specific instruction (instruction/cycle) [27]. The instruction descriptions are as follows:

- `UMULL` performs 2 unsigned 32×32 multiplications and produces a pair of 64-bit products.
- `ADD` (Adv. SIMD) performs 2 unsigned 64-bit additions.
- `LD4`, `ST4` load and store 4 128-bit vectors of data simultaneously.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 1: Instructions performance comparison for ARMv8 Cortex-A57 A64 and Adv. SIMD [27]

Architecture	Instruction	Execution Latency	Execution Throughput
Adv. SIMD	UMULL	5/4	1
	ADD	3	2
	LD4	11	1/4
	ST4	8	1/8
A64	MUL	3	1
	UMULH	6	1/4
	ADD	1	2
	LDP	4	1
	STP	1	1

- MUL performs an unsigned 64×64 multiplication and produces a single 64-bit register as the least significant half of the product.
- UMULH performs an unsigned 64×64 multiplication and produces a single 64-bit register as the most significant half of the product.
- ADD (A64) performs one unsigned 64-bit addition.
- LDP, STP load and store a pair of 64-bit general purpose registers simultaneously.

In the following section, we will present our implementation approach based on these instructions and their timings in more details.

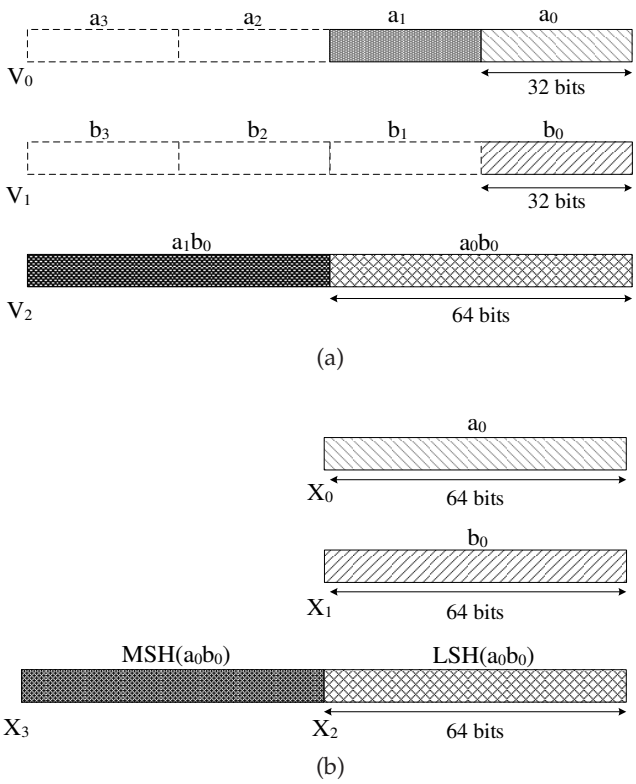

Figure 2: Multiplication using (a) Adv. SIMD and (b) A64 instructions.

4.1.1 ARMv8 A64 vs. Adv. SIMD Arithmetic Implementation

The Adv. SIMD instruction set and its capabilities in ARMv8 are very similar to ARMv7 NEON capabilities except for the number of vector registers. AArch64 provides $31\times$64-bit general-purpose registers as well as $32\times$128-bit vector registers which is almost twice as the number of available registers in ARMv7 platforms [19]. This large number of registers reduces data transfer operations between registers and memory significantly, and offer more efficient implementation of relatively-large finite field arithmetic.

ARMv8 Adv. SIMD is capable of computing multiplications over two pairs of 32-bit values to produce one pair of 64-bit products at a time. On the other hand, A64 multiplication instruction is capable of computing one pair of 64-bit products using two multiplication instructions, one for the least significant half (LSH) using MUL instruction, and one for the most significant half (MSH) using UMULH instruction. As a result, each 128-bit partial product can be computed using a single multiplication instruction in Adv. SIMD and two multiplication instructions using A64 general-purpose registers. However, regardless of the difference between latency and throughput of multiplication instructions in A64 and Adv. SIMD, as it is illustrated in Fig. 2, the computing word size in A64 general-purpose architecture is twice as Adv. SIMD registers. In contrast to ARMv7 32-bit platform where both A32 general-purpose registers and NEON vectorization architecture provide the same 2^{32}-radix representation of operands and taking advantage of parallel 32×32 multiplication of NEON instruction results in remarkable performance improvement in finite field arithmetic implementation [18], [28], [29].

The extra number of words in ARMv8 Adv. SIMD requires twice as many single-precision multiplication compared to A64 which has a considerable effect on performance. However, as it is discussed, since 128-bit partial products can be computed using single Adv. SIMD multiplication instruction, the total number of multiplication instructions for a multi-precision multiplication function will be the same for both designs. This fact makes the SIMD design comparable to A64 implementation; however, based on execution latencies in Table 1,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 2: Performance comparison (CPU clock cycles) of 751-bit finite field school-book multiplication on a single ARM Cortex-A57 core using A64 general registers vs. Adv. SIMD vectorization

Operation	**A64**	**Adv. SIMD**
Multiplication	1,041	1,363
Montgomery Reduction	627	820

the total cycle counts for two multiplication instructions in Adv. SIMD is more than the overall execution cycles for `MUL` and `UMULH` instructions on Cortex-A57 processors; thus, we remark that using A64 general-purpose registers should provide faster timing results compared to Adv. SIMD implementation on ARMv8 platforms. To confirm this claim, we implemented two versions of optimized 751-bit school-book multiplication and Montgomery reduction over $p751$ finite field in assembly using A64 and Adv. SIMD instruction sets. For Adv. SIMD implementation, we deployed the same strategy which was first introduced in [28] and deployed in [18] for SIDH implementation. The performance results on a Cortex-A57 core in Table 2 show that unlike ARM-NEON implementation on ARMv7 platforms, ARMv8 Adv. SIMD performance suffers from extra radix representation and does not provide any improvement compared to A64 design. Therefore, we choose to implement our ARMv8 optimized finite field arithmetic library using ARMv8 A64 assembly instructions, taking advantage of its wide 64-bit general-purpose registers.

4.2 Arithmetic in $\mathbb{F}_p$

Finite field arithmetic is the fundamental building block in number-theoretic cryptographic protocols. The SIDH protocol implementation works over quadratic extension fields $\mathbb{F}_{p^2}$ [15]. However, the arithmetic over this field is implemented based on finite field arithmetic over the base field $\mathbb{F}_p$. Thus, highly-optimized field arithmetic modulo p is necessary for developing SIDH fast implementation. The needed operations are finite field addition, squaring, multiplication, and inversion.

4.2.1 Finite Field Addition and Subtraction

For ARMv8, field addition can be implemented efficiently using A64 instructions since addition and subtraction instructions are capable of adding and subtracting 64-bit operands, respectively, using a single instruction.

The multi-precision field addition is implemented by loading operands into two 64-bit registers at a time using `LDP` instruction and adding the carry bit. For constant-time implementation, at the end of addition operations, the negative of the prime value is added to the result. Based on a bit-mask, the prime value or zero is re-added to this result again in order to correct the final result if it is larger than the prime value.

Similarly, the field subtraction is implemented using subtraction instruction with borrow. For constant-time implementation, the prime value is added to the result and, according to a bit-mask, the final result is computed.

4.2.2 Finite Field Multiplication

In this work, finite field multiplication is performed over two different fields. We adopt different approaches for computing finite field multiplication for each field, taking advantage of the special form of the primes.

751-bit multiplication. Each 751-bit element can be represented using 12×64-bit registers on 64-bit platforms. The large number of 64-bit registers on ARMv8 processors provides the capability of optimized arithmetic implementation over relatively large finite fields. Specifically, since data load instructions from memory into registers take a considerable amount of execution time on the ARM platform, compact implementation of multi-precision multiplication is desirable. To this end, we implemented 751-bit multiplication using Comba multiplication algorithm. We notice that the number of registers on ARMv8 processors is redundant enough to implement multiplication over this field only with a small number of data transfers between memory and registers. However, over larger finite fields, the Comba multiplication performance deprecates due to memory transfer instructions.

964-bit multiplication. 964-bit operands can be represented using 16×64-bit registers on ARMv8 processors. The main disadvantage of this representation is that the most significant word only contains 4-bit data which leads to considerable redundancy in the representation and thereby performance downgrading. However, in this section, we explain how to take advantage of this redundant representation to reduce the cost of carry propagation using two-level Karatsuba multiplication. A similar approach has been used before in [30] by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

merging "refined Karatsuba" multiplication with a subsequent modular reduction, targeting ARMv7 Cortex-A8 processors. However, their implementation benefits significantly from intermediate reduction over a Mersenne prime which cannot be utilized in our setting.

We decompose an integer a modulo $p964$ into 16 $\times$ 64-bit registers in mixed, yet symmetric radix representation. We represent a as $a_0 + 2^{64}a_1 + 2^{128}a_2 + 2^{192}a_3 + 2^{241}a_4 + 2^{305}a_5 + 2^{369}a_6 + 2^{433}a_7 + 2^{482}a_8 + 2^{546}a_9 + 2^{610}a_{10} + 2^{674}a_{11} + 2^{723}a_{12} + 2^{787}a_{13} + 2^{851}a_{14} + 2^{915}a_{15}$. Note that since Adv. SIMD vectorization does not provide any performance benefits on ARMv8 platforms, as it is illustrated in Fig. 3, we only set $a_3, a_7, a_{11},$ and a_{15} in redundant representation. These limbs contain 49 bits, while other limbs include 64 bits. With this decomposition, we are able to design our tailored two-level Karatsuba multiplication for 964-bit elements. Each limb $a_3, a_7, a_{11},$ and a_{15} is smaller than 64 bits and can be fit into ARMv8 general-purpose registers. Moreover, there is enough space available in these limbs that can be exploited to delay carry propagation operations in Karatsuba multiplication.

For the first level of Karatsuba multiplication, one 16-word integer a is divided into two 8-word integers A_0 and A_1, i.e., $a = A_0 + 2^{482}A_1$ as:

$$A_0 = a_0 + 2^{64}a_1 + 2^{128}a_2 + 2^{192}a_3 + 2^{241}a_4 + 2^{305}a_5 + 2^{369}a_6 + 2^{433}a_7;$$

$$A_1 = a_8 + 2^{64}a_9 + 2^{128}a_{10} + 2^{192}a_{11} + 2^{241}a_{12} + 2^{305}a_{13} + 2^{369}a_{14} + 2^{433}a_{15}.$$

We also decompose the other operand b applying the same strategy and we have:

$$a \cdot b = A_1B_1 2^{964} + [(A_0 + A_1) \cdot (B_0 + B_1) - A_1B_1 - A_0B_0]2^{482} + A_0B_0.$$

For the second level of Karatsuba multiplication, we split each 8 words of A_0 and A_1into two 4-limb integers $(A_{00}, A_{01}, A_{10}, A_{11})$. Now, we show $A_0 = A_{00} + 2^{241}A_{01}$ and $A_1 = A_{10} + 2^{241}A_{11}$ as:

$$A_{00} = a_0 + 2^{64}a_1 + 2^{128}a_2 + 2^{192}a_3;$$
$$A_{01} = a_4 + 2^{64}a_5 + 2^{128}a_6 + 2^{192}a_7;$$
$$A_{10} = a_8 + 2^{64}a_9 + 2^{128}a_{10} + 2^{192}a_{11};$$
$$A_{11} = a_{12} + 2^{64}a_{13} + 2^{128}a_{14} + 2^{192}a_{15}.$$

We apply the same decomposition for B_0 and B_1 to acquire B_{00}, B_{01}, B_{10}, and B_{11}. Now, we compute A_0B_0 and A_1B_1 as:

Table 3: Performance comparison (CPU clock cycles) of 964-bit finite field multiplication on a single ARM Cortex-A57 core

Algorithm	Cycles
Two-level Karatsuba	1,244
Comba	1,636

$$A_0B_0 = A_{01}B_{01}2^{482} + [(A_{00} + A_{01}) \cdot (B_{00} + B_{01}) - A_{01}B_{01} - A_{00}B_{00}]2^{241} + A_{00}B_{00},$$
$$A_1B_1 = A_{10}B_{10}2^{482} + [(A_{10} + A_{11}) \cdot (B_{10} + B_{11}) - A_{11}B_{11} - A_{10}B_{10}]2^{241} + A_{10}B_{10}.$$

On the lowest level, we compute 4-limb integer multiplication using Comba multiplication which can be efficiently implemented using A64 general-purpose registers on ARMv8 without extra memory transfer instructions.

To evaluate the performance of our proposed multiplication strategy, we also implemented 964-bit multiplication using Comba multiplication algorithm in ARM assembly and provided the timing results in Table 3. We verified that our tailored two-level Karatsuba multiplication performs almost 24% faster than Comba multiplication on our targeted processor.

4.2.3 Finite Field Reduction

We choose to use prime of the form $p = 2^{\alpha} \cdot 3^{\beta} - 1$ for our ARM-based software because of its efficient features. We deployed the same reduction technique mentioned in [16] instead of generic Montgomery [31] or Barret [32] reduction algorithms. The proposed technique is novel and yet straightforward to implement. That is, instead of computing the Montgomery residue $c = aR^{-1} \bmod p$ for an input $a < pR$, by using

$$c = \left(a + \left(ap' \bmod 2^R\right) \cdot p\right) / 2^R$$

which requires roughly $s^2 + s$ multiplications for a $2s$-limb value a in generic form, computations can be simplified to

$$c = \left(a + \left(ap' \bmod 2^R\right) \cdot 2^{\alpha} \cdot 3^{\beta} - \left(ap' \bmod 2^R\right)\right) / 2^R$$
$$= a/2^R + \left(\left(ap' \bmod 2^R\right) \cdot 3^{\beta}\right) / 2^{R-\alpha}$$

for $p = 2^{\alpha} \cdot 3^{\beta} - 1$. Based on the above formula, instead of multiplication with the prime value, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

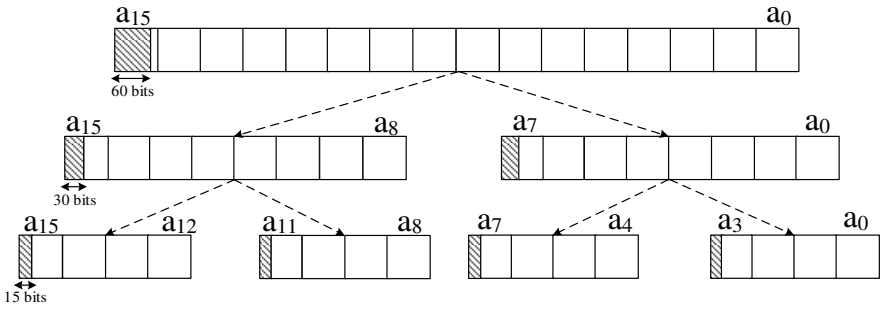

Figure 3: Two-level Karatsuba multiplication for 964-bit operands.

multiplication with $p+1 = 2^{\alpha} \cdot 3^{\beta}$ is computed which has exactly $\lfloor \frac{\alpha}{r} \rfloor$ least significant words equal to "0" for 2^r-radix representation of elements. For instance, $p751+1$ and $p964+1$ have five and seven least significant words equal to "0", respectively, which can simply be eliminated. Additionally, one can simply notice that the primes of this form are also *Montgomery-friendly* primes [33] which reduces the s^2+s multiplications to s^2 multiplications for a $2s$-limb value, since $p' = -p^{-1} \bmod 2^R$ is equal to 1 and s multiplications are eliminated. We implemented optimized Comba-based Montgomery reduction in product scanning form for both primes.

4.2.4 Finite Field Inversion

There exist different approaches to compute the inverse of an operand over a finite field. These approaches differ in complexity as well as security due to constant or non-constant implementations. Constant-time modular inversion algorithms are significantly slower than their non-constant counterparts, but since they are resistant against power analysis attacks, they are more desirable for cryptography applications. For instance, as it was pointed out in [34], using projective representation can reveal information about secret data during the conversion from projective to affine coordinates. Therefore, regardless of performance degradation, a constant-time inversion method should be adopted in projective settings.

One constant-time approach is Fermat's Little Theorem (FLT) which computes the field inversion based on $x^{-1} = x^{p-2} \bmod p$, using field multiplication and squaring in $O(\log^3 p)$. This method of computing inversions is computationally intensive, and it is mostly deployed when the use of field inversion is scarce inside the protocol since otherwise the performance degradation would be catastrophic. As an alternative, Bos's constant-time inversion algorithm [35] computes the inversion of an element based on the binary GCD without using any field multiplication or squaring.

The first projective based SIDH key exchange implementation [16] includes only one field inversion at the very end of each key exchange step. Moreover, inversion computation using addition chains significantly benefits from Montgomery arithmetic inside the protocol. Therefore, using constant-time FLT algorithm for inversion seems to be reasonable in their settings leading to only minor performance degradation. Nevertheless, since the proposed method in [35] also provided more efficient timings on ARM-powered devices compared to FLT, we explored the performance impact of using this approach in our projective SIDH software for both security levels. In particular, since ARM processors are not as resource-rich as Intel processors, even a small fraction of optimization is desirable.

We have implemented the binary GCD inversion algorithm using hand-written assembly instructions on our target processor. The proposed constant-time implementation in [35] includes multi-precision addition, subtraction, and shift operations over the finite field as well as simple logic operations on word-size operands. Since the proposed implementation is constant-time, the total number of $2\lceil \log_2(p) \rceil$ iterations are performed for every input. Based on the implementation results in [35], we expected to see more than 2 times faster inversion software compared to FLT algorithm. However, we observed significant difference in our implementation results on 64-bit ARM processor. Table 4 summarizes the performance cost on our benchmark platform over the 751-bit and 964-bit finite field inversion. We observe that using FLT algorithm provides faster results compared to binary GCD algorithm for both 751-bit and 964-bit primes. Note that these timing results are obtained using our highly-optimized arithmetic library for both inversion methods. Furthermore, for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 4: Performance numbers in 10^3 cycles for finite field inversion modulo 751-bit and 964-bit SIDH primes on ARM Cortex-A57

	Constant time		Non-constant time
Prime	FLT	Binary GCD	EEA
$p751$	1,368	1,792	31
$p964$	2,570	2,939	40

constructing addition chains inside FLT algorithm, we used 6-bit window method with precomputed table. In this case, the table includes only required exponents of an element a up to a^{63}. We address the difference between our timing results and the results presented in [35], regardless of the target platform, as in our implementation, we use highly-optimized multiplication and squaring functions as well as using Montgomery arithmetic for computing large exponentiation. Moreover, the presented results in [35] are based on binary method for computing exponentiation in FLT which requires almost b modular squarings and $\frac{b}{2}$ modular multiplications for generic b-bit prime moduli p, in contrast to our window-based implementation which includes much less modular multiplications and thereby is more efficient.

In contrast to projective SIDH formulas, affine SIDH software requires hundreds of modular inversions for each step of the key exchange protocol. As a result, the only comparative implementation of affine SIDH software can be developed using a non-constant time inversion method as it is used in [15], [17], and [18]. Among different non-constant time methods of computing modular inversion, the Extended Euclidean Algorithm (EEA) can be deployed to compute the inverse of an operand at a significantly lower time complexity of $O(\log^2 p)$ compared to FLT, but with leaking some information about the value being inverted from simple power analysis and timing attacks [36]. However, to provide some level of protections against timing attacks in affine SIDH software, a random value can be multiplied to the operand before and after the inversion. This requires two extra modular multiplications, but the additional defense against timing attacks necessitates this minor performance degradation. Like the previous versions of affine SIDH implementations, we also choose to use EEA method inside our affine SIDH software for modular inversions. The EEA implementation deploys our optimized arithmetic library for multiplication and reduction besides the GNU multi-precision library for computing the inverse of an element. We included the performance cost of this method on our benchmark platform in Table 4 for both finite fields.

Since the proposed primes in this work have the number of bits smaller than the multiple of 64-bit word, we adopt a combination of Karatsuba multiplication, carry-handling elimination, and lazy reduction in extension field arithmetic for achieving better performance similar to [16] and [37].

5 Affine *vs.* Projective SIDH

This section compares two different approaches of implementing SIDH protocol, namely the projective isogeny formulas presented by Costello et al. [16] and affine isogeny formulas introduced by De Feo et al. [15]. From the security point of view, the most significant distinction between these two formulas is that the projective isogeny provides constant-time, almost inversion-free point and isogeny arithmetic. However, in terms of performance, we need to look at the relative cost of an inversion which is used in the affine formulas and compare it with the cost of additional multiplications needed for projective formulas. To this end, we use the inversion/multiplication ratio, denoted as $R_p = I_p/M_p$ over $\mathbb{F}_p$ and $R_{p^2} = I_{p^2}/M_{p^2}$ over $\mathbb{F}_{p^2}$. The R_p ratio indicates the cost of an inversion to the cost of a multiplication over $\mathbb{F}_p$, while R_{p^2} denotes the cost of an inversion to the cost of a multiplication over extension field $\mathbb{F}_{p^2}$. In the case of constant-time inversion such as FLT for generic b-bit prime moduli p, using the addition chains method, the R_p ratio is equal to several hundreds and almost a thousand over our 751-bit and 964-bit prime fields, respectively; in contrast to non-constant time inversion algorithms like EEA, in which the cost of an inversion to the cost of a multiplication is significantly smaller. Since the only required inversion in SIDH protocol is over the quadratic extension field $\mathbb{F}_{p^2}$, the R_{p^2} ratio should be taken in consideration.

Let $\beta = b_0 + b_1\alpha \in \mathbb{F}_{p^2}$ be a non-zero element, where $\alpha^2 = \gamma \in \mathbb{F}_p$. The multiplicative inverse of β can be computed as follows:

$$\beta^{-1} = \frac{1}{b_0 + b_1\alpha} = \frac{b_0 - b_1\alpha}{b_0^2 - b_1^2\gamma} = \frac{b_0}{b_0^2 - b_1^2\gamma} - \frac{b_1\alpha}{b_0^2 - b_1^2\gamma}.$$

The inverse of β can be computed in $I_{p^2} = I_p + 2M_p + M_\gamma + 2S_p + \text{sub}_p + \text{neg}_p$. Roughly, we can assume $I_{p^2} \leq I_p + 6M_p$. On the other hand, the cost of multiplication in the quadratic extension field is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 5: Field arithmetic timings over the $p751$ and $p964$ prime fields on a single core ARM Cortex-A57 (average over 10^4 operations in CPU cycles (cc))

		C				ASM			
Prime	Field	Add	Mul	Inv (EEA)	**R=I/M**	Add	Mul	Inv (EEA)*	**R=I/M**
$p751$	$\mathbb{F}_p$	53	2,966	31,592	**10.65**	46	1,636	31,592	**19.31**
	$\mathbb{F}_{p^2}$	645	10,560	40,205	**3.81**	130	6,101	38,623	**6.33**
$p964$	$\mathbb{F}_p$	74	4,499	40,425	**8.98**	61	2,312	40,425	**17.48**
	$\mathbb{F}_{p^2}$	728	15,563	59,541	**3.82**	167	7,647	55,979	**7.32**

* Inversion over $\mathbb{F}_p$ is implemented only using GMP library.

$M_{p^2} \geq 3M_p$, since $M_{p^2} = 3M_p + M_\gamma + 2\text{add}_p + 2\text{sub}_p$. Thus,

$$R_{p^2} = I_{p^2}/M_{p^2} \leq (I_p/3M_p) + 2 = R_p/3 + 2,$$

which implies that for large ratio R_p, the R_{p^2} ratio is almost equal to $R_p/3$. Note that relatively-small R_{p^2} ratios make affine formulas faster than projective formulas in some implementations and settings, while larger values of R_{p^2} indicate projective formulas outperform its counterpart. Based on [16], SIDH projective formulas show much better performance on `x86-64` Intel processors. However, the inversion to multiplication ratio on other platforms and architectures might lead to performance improvements for affine SIDH implementations. To investigate this possibility, we give detailed performance numbers for both base field and quadratic extension field modular arithmetic on our target platform. This includes the inversion to multiplication ratios for both fields. In Table 5, we provide performance in terms of cycle counts for the SIDH required modular field arithmetic over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$; results represent both generic C implementation using GMP[1] library and optimized assembly using our assembly library.

The last column in Table 5 shows the inversion to multiplication ratio cost for each implementation. As it is indicated in this table, the ratios are much smaller for the quadratic extension field than the base field and as we expected $R_{p^2} \leq R_p/3 + 2$.

Recall that the large values of R_{p^2} indicate that affine SIDH performance suffers from excessive number of inversions. Thus, the projective SIDH is expected to show more efficient results. Table 5 shows that the R_{p^2} ratio for our optimized implementation is almost twice as the C implementation for both finite fields. Notably, efficient implementation of finite field multiplication over $p964$ field increases the R_{p^2} ratio significantly which should translate to better speedup of using projective SIDH over $p964$ compared to $p751$ implementation.

Based on these observations, we claim that projective formulas performance benefits significantly from optimized implementation of finite field multiplication. Thus, comparison between affine and projective SIDH performance is directly related to the target platform and field multiplication implementation. In the next Section, we evaluate performance results of the SIDH protocol for both affine and projective implementations on ARM Cortex-A57 processor.

6 Implementation Results and Discussion

In this section, we present the performance results of both affine and projective SIDH key exchange protocol software on the high-performance ARM Cortex-A57 processor. We use the exact same optimized field library for both affine and projective implementations over each finite field. Moreover, we set curve parameters $a = 0$ and $b = 1$ to construct the elliptic curve $E/\mathbb{F}_{p^2} : y^2 = x^3 + x$ as the starting Montgomery curve.

To evaluate the performance of the SIDH key exchange protocol, both affine and projective codes are compiled using the standard operating system on the Juno ARM Development Platform. The software is compiled with Linaro GCC v4.9.4 on a single core 1.1GHz ARM Cortex-A57 running OpenEmbedded Linux v4.5.0. Results represent the average of 10^3 iterations, reported in clock cycles to ease comparison. We measured CPU time and scaled the number to clock cycles using the processor frequency; accordingly, the cycle counts reported here represent an upper bound on the actual execution time.

1. GNU MP Bignum Library

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 6: Performance results ($\times 10^6$ CPU clock cycles) of affine and projective coordinates for SIDH key exchange protocol on different platforms for various quantum security level.

Work	Lang.	Device	Field size	PQ Security	Coordinate	Timings [cc $\times 10^6$]				
						Alice R1	Bob R1	Alice R2	Bob R2	Total
Costello et al. [16]	ASM	Haswell	751	125	Proj.	51	59	47	57	214
		Sandy Bridge				54	64	51	61	230
Koziel et al. [18]	C	Cortex-A15	751	125	Affine	437	474	346	375	1,632
	ASM		1008	167		603	657	516	484	2,259
Azarderakhsh et al. [17]	C	Cortex-A15	771	128	Affine	N/A	N/A	N/A	N/A	3,009
			1035	170		N/A	N/A	N/A	N/A	6,477
This Work	ASM	Cortex-A57	751	125	Affine	132	143	106	**112**	493
					Proj.	**103**	**118**	**97**	113	**431**
			964	160	Affine	276	291	223	**231**	1,021
					Proj.	**201**	**226**	**188**	233	**848**
	C		751	125	Affine	210	225	168	180	783
					Proj.*	568	671	528	633	2,400
			964	160	Affine	436	455	350	365	1,606
					Proj.	1,323	1,502	1,230	1,421	5,476

* The main portable C code from [16] is evaluated on ARM Cortex-A57 processor. The radix was changed to 2^{64} on ARM paltform.

6.1 Benchmark Results

Table 6 includes the benchmark results of our affine and projective software over two security levels. As we expected, the optimized version of projective formulas computes the key exchange protocol faster than affine formulas over both finite fields. However, we notice that affine software slightly outperforms its projective counterparts for the final round of the protocol and show better performance. Furthermore, the difference between projective and affine formulas are more dominant over $p964$ finite field as we expected based on our I/M analysis. Timing results indicate that the optimized projective SIDH performs almost 14% and 20% faster than affine SIDH over $p751$ and $p964$ fields, respectively.

6.2 Comparison

Since this work provides efficient implementation of the SIDH protocol on ARMv8 processors, we compare our results with previous implementations on ARM-powered devices at the same level of security. The only other publicly available optimized implementations of SIDH are [17], [16], [18] which only provide generic implementations on ARMv8 platforms. As we expected, our optimized software performs much better than generic implementations. In particular, although the optimized SIDH projective implementation of [16] only targets `x86_64` Intel processors, we compiled their generic implementation on our target platform for comparison, with the understanding that their portable version is not optimized for our platform.

If we compare our implementation to that of [16] on different target platforms, our optimized implementations are 2 times slower in terms of cycle counts, mainly because of differences between ARM and Intel processor architectures. ARM processors are based on RISC (Reduced Instruction Set Computing) architecture which focuses on power-efficiency, while Intel processors are based on CISC (Complex Instruction Set Computing), making direct cross-architecture comparisons of limited utility.

In comparison with other ARM-based implementations, our software is the fastest SIDH key exchange implementation for a given post-quantum security level.

6.3 Open Quantum Safe Benchmark

In this section, we compare other quantum-resistant cryptographic primitives with SIDH in terms of performance, security and communication overhead. Bos et al. [41] recently proposed a new quantum-resistant primitive based on lattices. They compared the implementation metrics of their proposed key exchange protocol with other quantum-resistant primitives on the `x86_64` architecture.

To evaluate the efficiency of our software, we integrated our optimized implementation of projective SIDH key exchange into both liboqs and Open Quantum Safe OpenSSL projects [42]. The OpenSSL

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2723891, IEEE Transactions on Dependable and Secure Computing

Table 7: Performance evaluation of post-quantum cryptography key exchange protocols. All the results are generated by Open Quantum Safe OpenSSL library, compiled and evaluated on a single core ARMv8 Cortex-A57 processor

Protocol	Lang.	Alice0 (ms)	Bob (ms)	Alice1 (ms)	Communication (bytes) $A \rightarrow B$	Communication (bytes) $B \rightarrow A$	PQ Security
RLWE BCNS [38]	C	2.85	4.65	0.695	4,096	4,224	76
RLWE NewHope [39]	C	0.284	0.442	0.106	1,824	2,048	206
RLWE MSR [40]	C	0.199	0.361	0.065	1,824	2,048	206
LWE Frodo Recomm. [41]	C	59.3	59.9	0.427	11,280	11,288	130
SIDH [16]	C	497	1114	468	564	564	125
SIDH (This Work)	**C/ASM**	**97**	**216**	**90**	**564**	**564**	**125**
	C/ASM	**178**	**376**	**183**	**726**	**726**	**160**

library is compiled with Linaro GCC v4.9.4 on a single core 1.1GHz ARM Cortex-A57 running OpenEmbedded Linux v4.5.0. We are following the same strategy in [41] and [38] as comparing performance of standalone cryptographic operations with other post-quantum cryptosystem candidates on our targeted platform. Table 7 demonstrates the overall comparison of different post-quantum cryptosystems. In this table, **Alice0** shows Alice's key and message generation operations; **Bob** denotes Bob's key and message generation and his shared key computations; and finally **Alice1** represents the final shared key computations by Alice.

Our optimized software demonstrates the speedup of up to 5X over the generic C implementation of SIDH [16] for the same security level. However, note that the large number of operations to calculate the isogeny map between elliptic curves, even inside our optimized implementation of SIDH, affects the overall performance of this scheme and makes it slower compared to its counterparts. However, significantly smaller key size makes the SIDH key exchange protocol suitable for the applications where the communication overhead is a concern.

Moreover, most of the other post-quantum primitives are based on the hardness of lattice problem on ideal lattices and as it is stated in [41], recent cryptanalysis efforts show that their underlying security *might* be influenced. Nonetheless, RLWE-based cryptosystems show remarkable results in terms of performance.

7 Conclusion

In this paper, we have presented two optimized SIDH key exchange implementations for two different quantum security levels on the ARMv8 platforms. Our implementations provided both 125-bit and 160-bit quantum security. We investigated different implementation approaches on ARMv8 processors based on its architecture capabilities. Our field arithmetic library computes field-level SIDH operations faster than all other prior implementations on ARM platforms found in the literature. We introduced a new implementation-friendly prime for higher security level implementation of SIDH on ARM platforms. Moreover, we provided a comprehensive comparison between affine and projective SIDH formulas based on inversion-to-multiplication ratio, and concluded that the optimized projective SIDH always shows better performance compared to optimized affine SIDH on ARMv8 platforms.

We integrated our optimized software into open quantum safe OpenSSL project to compare its overall performance with other post-quantum cryptography primitives. Our benchmarked results demonstrate that although SIDH key exchange protocol shows slower timings compared to RLWE-based primitive, its significantly smaller key size makes this scheme suitable for the applications where the communication bandwidth is restricted.

We remark that since isogeny-based cryptosystems are younger than other post-quantum cryptography candidates, their performance and secu-

rity are still required to be studied widely. Nevertheless, the key size and performance of our software demonstrate the strong potential of SIDH as a quantum-resistant key exchange candidate. We hope that this work would be a paradigm shift towards motivating more investigation in this area.

Acknowledgment

The Authors would like to thank the reviewers for their constructive comments. This work was supported by NSF grant no. CNS-1661557 and NIST award 60NANB16D246.

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. IEEE Symp. Foundations of Computer Science*, 1994, pp. 124–134.

[2] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," *National Institute of Standards and Technology Internal Report*, vol. 8105, 2016.

[3] R. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.

[4] J. Patarin, "Hidden fields equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms," in *Proc. Int. Conf. on the Theory and Applications of Cryptographic Techniques*, 1996, pp. 33–48.

[5] R. C. Merkle, R. Charles *et al.*, "Secrecy, authentication, and public key systems," *Ph.D. thesis, Stanford University*, 1979.

[6] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Proc. Int. Algorithmic Number Theory Symp.*, 1998, pp. 267–288.

[7] A. Rostovtsev and A. Stolbunov, "Public-key cryptosystem based on isogenies," *IACR Cryptology ePrint Archive*, vol. 2006, p. 145, 2006.

[8] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. Int. Workshop on Post-Quantum Cryptography*, 2011, pp. 19–34.

[9] C. Delfs and S. D. Galbraith, "Computing isogenies between supersingular elliptic curves over $\mathbb{F}_p$," *Designs, Codes and Cryptography*, vol. 78, no. 2, pp. 425–440, 2016.

[10] J.-F. Biasse, D. Jao, and A. Sankar, "A quantum algorithm for computing isogenies between supersingular elliptic curves," in *Proc. Int. Conf. in Cryptology in India*, 2014, pp. 428–442.

[11] S. Zhang, "Promised and distributed quantum search," in *Proc. Int. Computing and Combinatorics Conf.*, 2005, pp. 430–439.

[12] S. Tani, "Claw finding algorithms using quantum walk," *Theoretical Computer Science*, vol. 410, no. 50, pp. 5285–5297, 2009.

[13] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proc. ACM Int. Workshop on ASIA Public-Key Cryptography*, 2016, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2898420.2898421.

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

[15] L. De Feo, D. Jao, and J. Plût, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *J. Mathematical Cryptology*, vol. 8, no. 3, pp. 209–247, 2014.

[16] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie-Hellman," in *Proc. Advances in Cryptology*, 2016, pp. 572-601.

[17] R. Azarderakhsh, D. Fishbein, and D. Jao, "Efficient implementations of a quantum-resistant key-exchange protocol on embedded systems," Technical report, 2014, http://cacr.uwaterloo.ca/techreports/2014/cacr2014-20.pdf.

[18] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani, "NEON-SIDH: Efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM," in *Proc. Int. Conf. on Cryptology and Network Security*, 2016, pp. 88–103.

[19] ARM Limited, "ARM — Q2 2016 Earnings Release," 2016, available at http://ir.arm.com/.

[20] J. Silverman, "The Arithmetic of Elliptic Curves 2nd edition, volume 106 of Graduate Text in Mathematics," 2009.

[21] J. Tate, "Endomorphisms of abelian varieties over finite fields," *Inventiones mathematicae*, vol. 2, no. 2, pp. 134–144, 1966.

[22] R. Bröker, "Constructing supersingular elliptic curves," *J. Comb. Number Theory*, vol. 1, no. 3, pp. 269–273, 2009.

[23] J. Vélu, "Isogénies entre courbes elliptiques," *CR Acad. Sci. Paris Sér. AB*, vol. 273, pp. A238–A241, 1971.

[24] J. W. Bos and S. Friedberger, "Fast Arithmetic Modulo $2^x p^y \pm 1$," Cryptology ePrint Archive, Report 2016/986, 2016, http://eprint.iacr.org/2016/986.

[25] Supersingular Isogeny-based Diffie-Hellman Key Exchange (SIDH) software v2.2, 2017, https://github.com/Microsoft/PQCrypto-SIDH.

[26] ARMv8 library for compressing and decompressing SIDH keys, 2016, http://csclub.uwaterloo.ca/~dburbani/work/keycomp-aug24-2016_final.zip.

[27] 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.

[28] H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim, "Montgomery Modular Multiplication on ARM-NEON Revisited," in *Proc. Int. Conf. on Information Security and Cryptology*, 2014, pp. 328–342.

[29] H. Seo, Z. Liu, J. Großschädl, and H. Kim, "Efficient Arithmetic on ARM-NEON and Its Application for High-Speed RSA Implementation." IACR Cryptology ePrint Archive, 2015, https://eprint.iacr.org/2015/465.pdf.

[30] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, "Curve41417: Karatsuba revisited," in *Proc. Int. Workshop on Cryptographic Hardware and Embedded Systems*, 2014, pp. 316–334.

[31] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[32] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory and Application of Cryptographic Techniques*, 1986, pp. 311–323.

[33] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Engineering*, vol. 5, no. 2, pp. 141–151, 2015.

[34] D. Naccache, N. P. Smart, and J. Stern, "Projective coordinates leak," in *Proc. Int. Conf. on the Theory and Applications of Cryptographic Techniques*, 2004, pp. 257–267.

[35] J. W. Bos, "Constant time modular inversion," *J. Cryptographic Engineering*, vol. 4, no. 4, pp. 275–281, 2014.
[36] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. Annual Int. Cryptology Conf.*, 1996, pp. 104–113.
[37] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López, "Faster explicit formulas for computing pairings over ordinary curves," in *Proc. Annual Int. Conf. on the Theory and Applications of Cryptographic Techniques*, 2011, pp. 48–68.
[38] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the tls protocol from the ring learning with errors problem," in *Proc. IEEE Symp. on Security and Privacy*, 2015, pp. 553–570.
[39] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange-a new hope," Cryptology ePrint Archive, Report 2015/1092, Tech. Rep., 2015, http://eprint.iacr.org.
[40] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," Cryptology ePrint Archive, Report 2016/504, 2016, http://eprint.iacr.org/2016/504.
[41] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the ring! practical, quantum-secure key exchange from lwe," in *Proc. ACM SIGSAC Conf. on Computer and Communications Security*, 2016, pp. 1006–1018.
[42] D. Stebila and M. Mosca, "Post-quantum key exchange for the internet and the open quantum safe project," Cryptology ePrint Archive, Report 2016/1017, Tech. Rep., 2016, http://eprint.iacr.org/2016/1017.

Amir Jalali received the B.Sc. degree in Computer Engineering from Shahid Beheshti University, Tehran, Iran in 2009, and the M.Sc. in Computer Engineering from the Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran in 2012. He is a Ph.D. candidate in Computer Engineering at the Department of Computer, Electrical Engineering and Computer Science at Florida Atlantic University, Boca Raton, FL, USA. Currently, he is a research intern in Cryptography Research Group at Microsoft Research, Redmond, WA. His current research interests include efficient software implementation of elliptic curve cryptography and post-quantum cryptography.

Reza Azarderakhsh (M'12) received 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 in the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. Currently, he is an assistant professor at the Department of Electrical and Computer Engineering at Florida Atlantic University. His current research interests include finite field and its application, elliptic curve cryptography, pairing based cryptography, and post-quantum cryptography. He is serving as an Associate Editor of IEEE Transactions on Circuits and Systems (TCAS-I).

Mehran Mozaffari Kermani (S'00-M'11-SM'16) received the B.Sc. degree from the University of Tehran, Iran, and the M.E.Sc. and Ph.D. de- grees from the University of Western Ontario, London, Canada, in 2007 and 2011, respectively. In 2012, he joined the Electrical En- gineering Department, Princeton University, New Jersey, as an NSERC post-doctoral research fellow. From 2013-2017 he was an Assistant Professor with Rochester Institute of Technology and starting 2017, he has joined the Computer Science and Engineering Department of University of South Florida. Currently, he is serving as an Associate Editor for the IEEE TVLSI, ACM TECS, and IEEE TCAS I. He has been the Guest Editor for the IEEE TDSC, IEEE/ACM TCBB, and the IEEE TETC for special issues on security. He has been the TPC member for a number of conferences including HOST (Publications Chair), DAC, DATE, RFIDSec, LightSec, WAIFI, FDTC, and DFT. He is a Senior Member of the IEEE.

David Jao received the Ph.D. degree in mathematics from Harvard University, Cambridge, MA, USA, in 2003. From 2003 to 2006, he worked in the Cryptography and Anti-Piracy Group at Microsoft Research, contributing cryptographic software modules for several Microsoft products. He is currently an associate professor in the Mathematics Faculty at the University of Waterloo, Canada, and the director of the Centre for Applied Cryptographic Research. His research interests include elliptic curve cryptography, protocol design and implementation, and post-quantum cryptography.