SIKE in 32-bit ARM Processors Based on Redundant Number System for NIST Level-II

HWAJEONG SEO, College of IT Engineering at Hansung University, Seoul, Republic of Korea
PAKIZE SANAL and REZA AZARDERAKHSH, Department of Computer, Electrical Engineering and Computer Science at Florida Atlantic University, Boca Raton, FL, USA

We present an optimized implementation of the post-quantum Supersingular Isogeny Key Encapsulation (SIKE) for 32-bit ARMv7-A processors supporting NEON engine (i.e., SIMD instruction). Unlike previous SIKE implementations, finite field arithmetic is efficiently implemented in a redundant representation, which avoids carry propagation and pipeline stall. Furthermore, we adopted several state-of-the-art engineering techniques as well as hand-crafted assembly implementation for high performance. Optimized implementations are ported to Microsoft SIKE library written in "a non-redundant representation" and evaluated in high-end 32-bit ARMv7-A processors, such as ARM Cortex-A5, A7, and A15. A full key-exchange execution of SIKEp503 is performed in about 109 million cycles on ARM Cortex-A15 processors (i.e., 54.5 ms @2.0 GHz), which is about 1.58× faster than previous state-of-the-art work presented in CHES'18.

CCS Concepts: • **Security and privacy → Public key (asymmetric) techniques**; • **Mathematics of computing** → *Mathematical software*; • **Computer systems organization** → *Embedded software*;

Additional Key Words and Phrases: Post quantum cryptography, software implementation, SIDH, ARM, parallel computation

ACM Reference format:
Hwajeong Seo, Pakize Sanal, and Reza Azarderakhsh. 2021. SIKE in 32-bit ARM Processors Based on Redundant Number System for NIST Level-II. *ACM Trans. Embed. Comput. Syst.* 20, 3, Article 19 (March 2021), 23 pages.
https://doi.org/10.1145/3439733

1 INTRODUCTION

Hard problems of traditional public key cryptography (e.g., RSA and Elliptic Curve Cryptography (ECC)) can be easily solved by using Shor's algorithm [Shor 1994] on an emerging quantum computer. For this reason, traditional public key cryptography cannot be secure anymore against

This work was partly supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (<Q|Crypton>, No. 2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity) and this work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1F1A1048478). The work of Reza Azarderakhsh is supported in part by NSF CNS-1801341 and NIST60NANB16D246.
Author's addresses: H. Seo (corresponding author), Hansung University, Republic of Korea; email: hwajeong84@gmail.com; P. Sanal and R. Azarderakhsh, Department of Computer, Electrical Engineering and Computer Science at Florida Atlantic University, Boca Raton, FL, USA; emails: {psanal2018, razarderakhsh}@fau.edu.

1539-9087/2021/03-ART19 $15.00
https://doi.org/10.1145/3439733

quantum attacks. A number of post-quantum cryptography algorithms have been proposed to resolve this issue. Among them, Supersingular Isogeny Diffie-Hellman key exchange (SIDH) protocol proposed by Jao and De Feo is considered as a premier candidate for post-quantum cryptosystems [Jao and Feo 2011]. Its security is believed to be secure against quantum algorithms on quantum computers. SIDH is the basis of the Supersingular Isogeny Key Encapsulation (SIKE) protocol [Azarderakhsh et al. 2019], which is currently under consideration by the National Institute of Standards and Technology (NIST) for inclusion in a future standard for post-quantum cryptography [NIST 2019]. Recently, NIST announced the round 3 finalist and alternatives and the SIKE is selected as an alternative candidate.[1] One of the attractive features of SIDH and SIKE protocols is their relatively small public key size, which are the most compact ones among well-established quantum-resistant algorithms. In spite of this prominent advantage, the "slow" speed of these protocols has been a sticking point, which hinders them to act like the practical solution. Therefore, the speeding up SIDH and SIKE protocols has become a critical issue as it judges the practicality of these isogeny-based cryptographic schemes in a real-world setting.

1.1 Efficient Implementations of SIDH and SIKE

The SIDH and SIKE implementations have been actively studied on both hardware and software. For the hardware implementation, efficient implementations of modular multiplication have been introduced. In Koziel et al. [2020], Montogmery multiplication and parallel computation based implementation was introduced. In Liu et al. [2019a], the SIDH hardware/software co-design implementation using the FFM2 hardware showed 31% faster than the best SIDH software implementation. In Liu et al. [2019b] a new high performance modular multiplication algorithm named HFFM for the specific fields in SIDH was proposed. This algorithm saved multiplications and additions compared with the previous algorithms.

For the software implementation, there are also several SIDH and SIKE implementations on various ARM embedded processors ranging from low-end ARM Cortex-M microcontrollers (for energy-efficient embedded devices) to high-end ARM Cortex-A processors (for supreme performance at optimal power). In 2018, the first implementation of SIDH on low-end 32-bit ARM Cortex-M4 microcontroller was suggested by Koppermann et al. [2018]. They utilized the state-of-the-art field arithmetic algorithm and optimize it in assembly language. An ephemeral key exchange (i.e., SIDHp751) on a 32-bit ARM Cortex-M4@120 MHz requires 18.833 s to perform—too slow to use on low-end microcontrollers. This work failed to prove the practicality of SIDH on low-end ARM microcontrollers. In CANS'19, the first practical SIKE result on a 32-bit ARM Cortex-M4 microcontroller was suggested by Seo et al. [2019a]. They presented new optimized implementation of modular arithmetic for the case of low-end 32-bit ARM Cortex-M4 microcontroller. With highly optimized modular arithmetic, the SIKE round 2 protocols for NIST Post Quantum Cryptography (PQC) competition (i.e., SIKEp434, SIKEp503, and SIKEp751) were efficiently implemented. The benchmark result on STM32F4 Discovery board equipped with 32-bit ARM Cortex-M4 microcontrollers shows that the entire key encapsulation over SIKEp434 takes only about 326 million clock cycles (i.e., 1.94 s @168 MHz). In Seo et al. [2020], the modular multiplication was further optimized by combining multiplication and reduction in a function. They presented all SIKE protocols such as SIKEp434, SIKEp503, SIKEp610, and SIKEp751. They achieved the 184 million clock cycles (i.e., 1.09 s @168 MHz). This is reasonably fast enough for practical applications.

For the high-end ARM processor, 32-bit ARMv7 Cortex-A for wearable devices and 64-bit ARMv8 Cortex-A for smartphones are widely used. First SIDH implementations on 64-bit ARMv8 Cortex-A processors are presented in 2017 [Jalali et al. 2017]. They analyzed the use of both affine

[1] https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement.

and projective SIDH formulas and provided a comprehensive analysis of both approaches based on the inversion-to-multiplication ratio. Benchmark results on ARMv8 demonstrated speedup of up to 5× over the generic version of SIDH implementation. In CHES'18, previous SIDH implementations on high-end 64-bit ARMv8 Cortex-A processors were improved further [Seo et al. 2018]. They presented high-speed Montgomery multiplication by utilizing 64-bit instructions and efficient computation orders. On an ARM Cortex-A72 from the ARMv8-A family, a full key-exchange execution of SIDHp503 was carried out in about 90 million cycles (i.e., 45 ms @1.992 GHz). In WISA'19, they optimized implementation of SIKE round 2 on 64-bit ARM, including SIKEp434 and SIKEp610 parameters [Seo et al. 2019b]. They used different optimization techniques to reduce the total number of underlying arithmetic operations on the field level. Benchmark results on the 64-bit ARM CortexA53@1.536 GHz processor showed that the entire SIKE round 2 key encapsulation mechanism took only 84 ms at NISTâs security level 1.

For the case of 32-bit ARMv7 Cortex-A processors, the first SIDH implementation was presented in CANS'16 [Koziel et al. 2016]. They provided fast affine SIDH implementations over 512, 768, and 1,024-bit primes on 32-bit ARM Cortex-A8 and A15 processors. The modular multiplication was optimized with the Cascade Operand Scanning (COS) method [Seo et al. 2016]. In SPACE'18, they presented a highly optimized implementation of SIKE mechanism on ARMv7 family of processors. They exploited state-of-the-art implementation techniques, including COS method and Karatsuba algorithm, and processor capabilities to efficiently develop post-quantum key encapsulation scheme on 32-bit ARMv7 Cortex-A processors [Jalali et al. 2018]. They achieved almost 7.5x performance improvement of the entire protocol over the SIKE 503-bit prime field on a Cortex-A8 core than reference code. In CHES'18, high-speed implementations of SIDH on 32-bit ARM Cortex-A15 processors were presented [Seo et al. 2018]. For the optimized multi-precision modular arithmetic, they finely integrated both ARM and NEON instructions, which reduces the number of pipeline stalls and memory accesses and presented a new Montgomery reduction technique that combines the use of the `UMAAL` instruction with a variant of the hybrid-scanning approach. On an ARM Cortex-A15 from the ARMv7-A family, a full key-exchange execution of SIDHp503 is carried out in about 176 million cycles (i.e., 88 ms @2.0GHz).

Previous optimized SIDH/SIKE implementations on 32-bit ARMv7 Cortex-A processors are based on a non-redundant representation. However, alternative approach, namely redundant representation, may achieve better performance than the non-redundant representation by referring previous ECC implementations on 32-bit ARMv7 Cortex-A processors. In CHES'12, the first cryptography implementation on 32-bit ARMv7 Cortex-A processors in a redundant representation was suggested [Bernstein and Schwabe 2012]. They showed ARM Cortex-A8 can sign a short message and verify a signature on a short message within 368,212 and 650,102 clock cycles for Ed25519, respectively. In CHES'14, the fast implementation of Elliptic Curve Diffie-Hellman (ECDH) over Curve41417 on 32-bit ARM Cortex-A8 was presented [Bernstein et al. 2014]. They utilized the redundant representation and refined Karatsuba algorithm. They achieved 1,648,409 clock cycles on the FreeScale i.MX515, which is faster than `secp160r1` of `openssl`. Previous works proved that well-designed redundant representation can lead to significant performance improvements.

1.2 Contribution

In this work, we improved SIKE implementations for NIST PQC competition (i.e., SIKEp503) on 32-bit ARMv7 Cortex-A processors. We present the new way to implement the finite field arithmetic in the redundant representation and Single Instruction Multiple Data (SIMD) instruction for SIKE parameters, which avoids carry propagation and pipeline stall. Furthermore, we adopted several state-of-the-art engineering techniques as well as hand-crafted assembly implementation. The optimized implementations are ported to Microsoft SIDH/SIKE library written in

"non-redundant representation" and evaluated in high-end 32-bit ARMv7 Cortex-A processors, such as ARM Cortex-A5, A7, and A15. A full key-exchange execution of SIKEp503 is performed in about 109 million cycles on 32-bit ARM Cortex-A15 processors (i.e., 54.5 ms @2.0 GHz), which is about 1.58× faster than previous state-of-the-art work presented in CHES'18.

1.3 Organization

This article is organized as follows. In Section 2, we briefly review the SIDH key exchange and SIKE key encapsulation mechanism. In Section 3.1, we introduce target high-end ARM processors (i.e., 32-bit ARMv7 Cortex-A) and previous implementations of modular multiplication on 32-bit ARMv7 Cortex-A processors. In Section 4, proposed implementations of finite field arithmetic are presented. Thereafter, we summarize our experimental results on 32-bit ARMv7 Cortex-A microcontroller in Section 5 and conclude the article in Section 6.

2 SUPERSINGULAR ISOGENY DIFFIE-HELLMAN KEY EXCHANGE

In this section, we briefly review SIDH and SIKE protocols and the required steps to generate a shared secret.

2.1 SIDH key Exchange

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

2.1.1 Public Parameters. Fix a prime p of the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, e_A and e_B are positive integers, and f is a very small cofactor. We define a based supersingular elliptic curve E_0 over $\mathbb{F}_{p^2}$ with cardinality $\#E_0 = (\ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \mp 1)^2$, and base points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ from the torsion subgroups $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$ respectively, such that $\langle P_A, Q_A\rangle = E_0[\ell_A^{e_A}]$ and $\langle P_B, Q_B\rangle = E_0[\ell_A^{e_B}]$.

2.1.2 Key Exchange Protocol. Alice randomly chooses two integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, not both divisible by ℓ_A as her secret key and computes an isogeny $\phi_A : E_0 \to E_A$ using kernel $R_A := \langle [m_A]P_A + [n_A]Q_A\rangle$. Alice also computes the image points $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$ by applying her secret isogeny ϕ_A to the public basis P_B and Q_B. She sends $\phi_A(P_B), \phi_A(Q_B)$ and E_A to Bob as her public key. Bob also selects random elements $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$, not both divisible by ℓ_B and computes a secret isogeny $\phi_B : E_0 \to E_B$ from kernel $R_B := \langle [m_B]P_B + [n_B]Q_B\rangle$, along with image points $\{\phi_B(P_A), \phi_B(Q_A)\} \subset E_B$. He sends his public key, i.e., $\phi_B(P_A), \phi_B(Q_A)$ and E_B to Alice.

In the second round of key exchange, Alice uses Bob's public key $(\phi_B(P_A), \phi_B(Q_A), E_B)$ and computes an isogeny $\phi'_A : E_B \to E_{AB}$ from kernel equal to $\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A)\rangle$. Similarly, Bob computes an isogeny $\phi'_B : E_A \to E_{BA}$ having kernel $\langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B)\rangle$ using Alice's public key. Since the common j-invariant of E_{AB} and E_{BA} are equal, they use this value to form a secret shared key. The entire SIDH key exchange protocol is illustrated in Figure 1.

2.2 SIKE Mechanism

SIKE mechanism is constructed by applying a transformation of Hofheinz et al. [2017] to the supersingular isogeny Public Key Encryption scheme described in Jao and Feo [2011]. It is an actively secure key encapsulation mechanism (IND-CCA KEM) that addresses the static key vulnerability of SIDH due to active attacks in Galbraith et al. [2016].

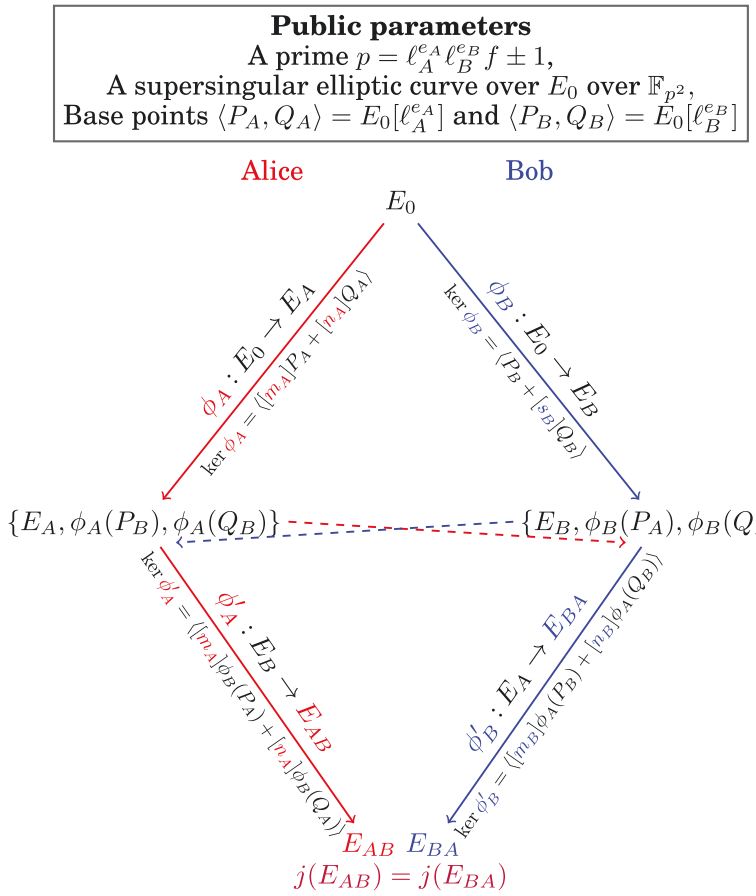

Fig. 1. SIDH key exchange protocol.

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

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

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

Encapsulation. Bob generates an t-bit random message $\text{m} \in_R \{0,1\}^t$, concatenates it with Alice's public key pk_A and computes an e_B-bit hash value r using cSHAKE256 hash function H_1, taking

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

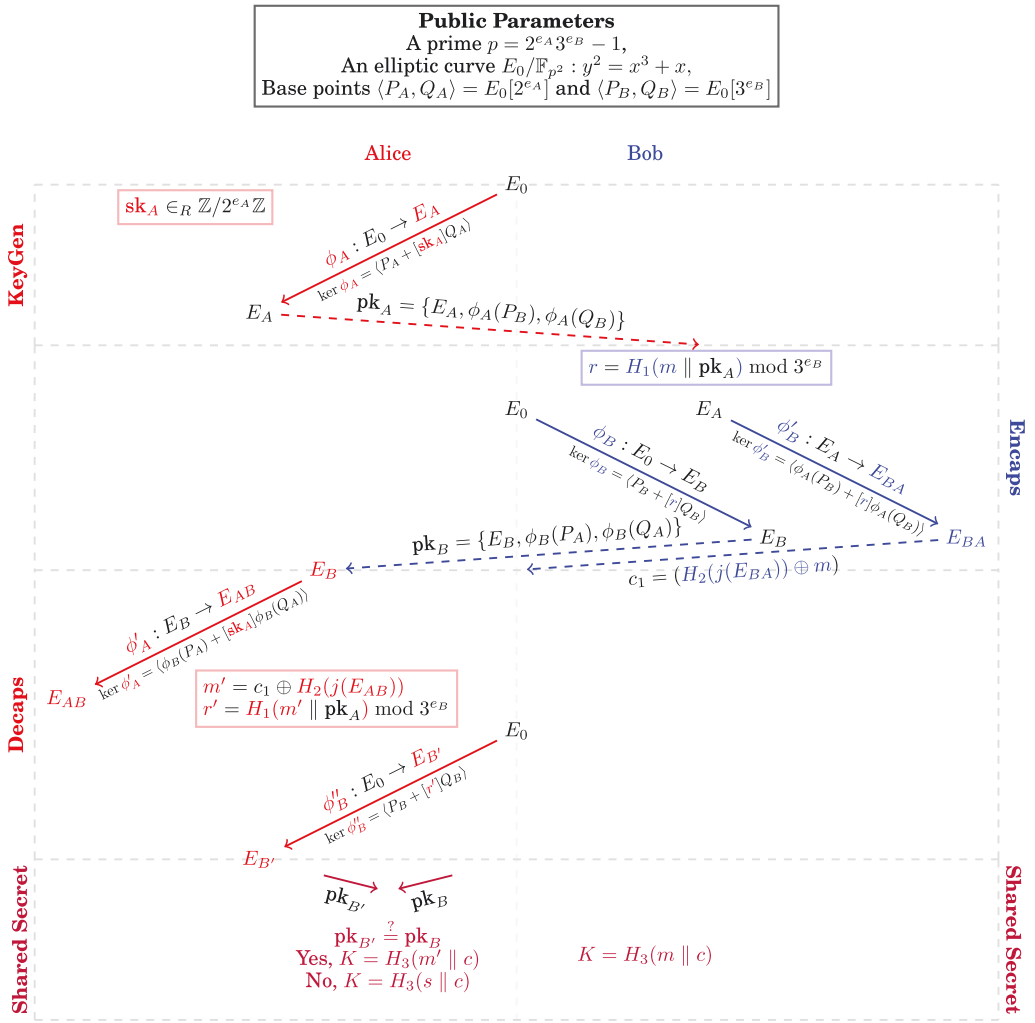

Fig. 2. SIKE mechanism.

$m \| \mathrm{pk}_A$ as the input. Using r, he applies a secret isogeny $\phi_B : E_0 \to E_B$ to the base points $\{P_A, Q_A\}$ and forms his public key $\mathrm{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$. Bob also computes the common j-invariant of curve E_{BA} by applying another isogeny $\phi'_B : E_A \to E_{BA}$ using Alice's public key. Bob forms a ciphertext $c = (c_0, c_1)$, such that:

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

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

Decapsulation. Upon receipt of c, Alice computes the common j-invariant of E_{AB} by applying her secret isogeny to E_B. She computes $m' = c_1 \oplus H_2(j(E_{AB}))$ and $r' = H_1(m \| \mathrm{pk}_A) \bmod 3^{e_B}$. Finally, she validates Bob's public key by computing $\mathrm{pk}_B(r')$ and comparing it with c_0. She generates the same shared secret $K = H_3(m' \| c)$ if the public key is valid; otherwise, she outputs a random value $K = H_3(c \| s)$ to be resistant against active attacks.

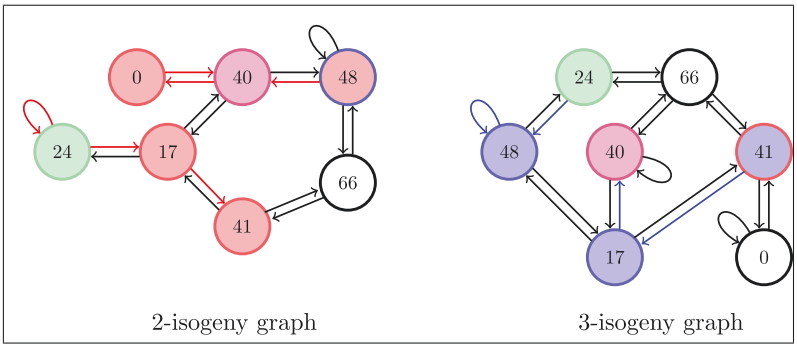

Fig. 3. Isogeny map in SIKE ($p = 71$, # of nodes $= 7 \approx p/12$, $j(E_0) = 24$, $j(E_A) = 41$, $j = (E_B) = j(E_{B'}) = 48$, $j(E_{AB}) = j(E_{BA}) = 40$).

As a toy example of SIKE, choose a prime $p = 2^3 \cdot 3^2 - 1 = 71$ and a starting elliptic curve $E_0 : y^2 = x^3 + x$ over $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with modulus $i^2 + 1 = 0$. Note that $j(E_0) = 24$. For this example, the isogeny maps between j-invariant classes are illustrated in Figure 3 as the number of supersingular j-invariants is $\approx p/12$. For the hash functions H_1, H_2, and H_3, we use a well-known hash function SHA-1. Choose the base points $\langle P_A, Q_A \rangle = E_0[2^3]$ and $\langle P_B, Q_B \rangle = E_0[3^2]$ as

$$\begin{aligned} P_A &= (5i + 26, 63i + 66), \\ Q_A &= (66i + 45, 66i + 8), \\ P_B &= (46i + 60, 39i + 70), \\ Q_B &= (30i + 41, 11i + 29). \end{aligned}$$

Suppose Alice picks $\mathrm{sk}_A = 3$ as her secret key and Bob picks a message $m =$ '*qwkhizoncl*'. Alice computes the isogeny $\phi_A : E_0 \to E_A$ (by following the 2-isogeny map path $24 \to 24 \to 17 \to 41$) so that

$$\begin{aligned} E_A &: y^2 = x^3 + 31x + 28, \\ \phi_A(P_B) &= (10i + 16, 15i + 20), \\ \phi_A(Q_B) &= (62i + 68, 62i + 24), \end{aligned}$$

and then she shares those as her public key with Bob. Bob calculates r by using his secret message m and Alice's public key pk_A:

$$r = H_1(m \parallel \mathrm{pk}_A) \bmod 3^2 = 0.$$

Then, Bob computes the isogeny $\phi_B : E_0 \to E_B$ (by following the 3-isogeny map path $24 \to 48 \to 48$) so that

$$\begin{aligned} E_B &: y^2 = x^3 + (28i + 10)x + (31i + 29), \\ \phi_B(P_A) &= (5i + 8, 3i + 48), \\ \phi_B(Q_A) &= (45i + 5, 62i + 48), \end{aligned}$$

and then he shares those values as his public key. He also computes the isogeny $\phi'_B : E_A \to E_{BA}$ (by following the 3-isogeny map path $41 \to 17 \to 40$) so that

$$E_{BA} : y^2 = x^3 + (26i + 15)x + (9i + 13),$$

and then he uses the $j(E_{BA})$ and m to compute

$$\texttt{0xaf3e133428b9e25c55bc2889382a219ccfae9217}$$

as c_1. He also shares c_1. Notice that Bob computes the shared key K,

$$\texttt{0xfcc1ddb6e9bc283c6e107355103952ce6b6639ae}.$$

Alice computes the isogeny $\phi'_A : E_B \to E_{AB}$ (by following the 2-isogeny map path $48 \to 40 \to 0 \to 40$) so that

$$E_{AB} : y^2 = x^3 + (26i + 15)x + (9i + 13).$$

Alice then obtains m' by using c_1 and $j(E_{AB})$, and therefore she obtains r' by using m' and Alice's public key pk_A,

$$m' = c_1 \oplus H(j(E_{AB})) = qwkhizoncl,$$
$$r' = H(m' \parallel \text{pk}_A) \bmod 3^2 = 0.$$

She finally computes the isogeny $\phi''_B : E_0 \to E_{B'}$ (by following 3-isogeny map path $24 \to 48 \to 48$) so that

$$E_{B'} : y^2 = x^3 + (28i + 10)x + (31i + 29),$$
$$\phi_{B''}(P_A) = (5i + 8, 3i + 48),$$
$$\phi_{B''}(Q_A) = (45i + 5, 62i + 48).$$

After checking whether those values are as same as Bob's public key, Alice computes the secret key K

$$\texttt{0xfcc1ddb6e9bc283c6e107355103952ce6b6639ae}.$$

It is easily seen that Alice's and Bob's shared secret keys are the same.

3 TARGET PROCESSOR AND MODULAR MULTIPLICATION

3.1 32-bit ARMv7 Cortex-A

In this article, we implemented proposed implementation methods on 32-bit ARMv7-A processors, which are widely used in mini–computers and wearable devices.

The target architecture supports both ARM and NEON instruction sets. On one hand, there are 16 32-bitwise ARM general purpose registers. Among these ARM registers, the engineer can utilize 14 32-bit registers, including `R0`∼`R12` and `R14` in assembly language. The ARM register can maintain 448-bit (32×14)-wise data.

However, the NEON engine provides 64-bit double (`D`) word and 128-bit quadruple (`Q`) word registers. For example, one 128-bit (`Q0`) register can be divided in two 64-bit (`D0` and `D1`) registers and manged. Such engineers can utilize sixteen 128-bit registers (or thirty-two 64-bit registers), including `Q0`∼`Q15` (or `D0`∼`D30`) in assembly language. The NEON register can retain 2,048-bit (128×16)-wise data, which is enough space to maintain long length of intermediate result and operand. Another feature of NEON instruction is a vectorized computation (i.e., 8-bit, 16-bit, 32-bit, and 64-bitwise), which utilizes a data-parallelism in instruction set level. For the 32-bit unsigned / signed vectorized multiplication, the NEON engine utilizes six multiplication instruction sets as follows:

- `VMULL.U32/S32` (Vectorized Unsigned / Signed Multiplication)
  `VMULL.U32 Q0, D2, D3[0]`
  → `D1 = D2[1]` × `D3[0], D0 = D2[0]` × `D3[0]`,

- `VMLAL.U32/S32` (Vectorized Unsigned / Signed Multiplication Accumulation)
  `VMLAL.U32 Q0, D2, D3[0]`
  → `D1 = D1 + D2[1] × D3[0], D0 = D0 + D2[0] × D3[0],`
- `VMLSL.U32/S32` (Vectorized Unsigned / Signed Multiplication and Subtraction)
  `VMLSL.U32 Q0, D2, D3[0]`
  → `D1 = D1 - D2[1] × D3[0], D0 = D0 - D2[0] × D3[0],`
- `VQDMULL.U32/S32` (Vectorized Unsigned / Signed Doubled Multiplication)
  `VQDMULL.U32 Q0, D2, D3[0]`
  → `D1 = 2 × D2[1] × D3[0], D0 = 2 × D2[0] × D3[0],`
- `VQDMLAL.U32/S32` (Vectorized Unsigned / Signed Doubled Multiplication Accumulation)
  `VQDMLAL.U32 Q0, D2, D3[0]`
  → `D1 = D1 + 2 × D2[1] × D3[0],`
  `D0 = D0 + 2 × D2[0] × D3[0],`
- `VQDMLSL.U32/S32` (Vectorized Unsigned / Signed Doubled Multiplication and Subtraction)
  `VQDMLSL.U32 Q0, D2, D3[0]`
  → `D1 = D1 - 2 × D2[1] × D3[0],`
  `D0 = D0 - 2 × D2[0] × D3[0].`

Instruction sets can issue two 32-bit unsigned multiplications and output two 64-bit results in parallel. However, unsigned instructions are usually useful for the non-redundant representation since the redundant representation needs to handle a sign bit. The signed alternative can be performed by replacing `U32` to `S32` option. Furthermore, multiplication results are efficiently doubled while computations with `VQDMULL`, `VQDMLAL`, and `VQDMLSL` instructions without additional doubling costs. These instructions are useful for some partial product calculations in the squaring operation.

In this work, we perform the benchmark of the proposed implementation on three 32-bit ARMv7 Cortex-A processors, including ARM Cortex-A5, A7, and A15. The ARM Cortex-A5 processor is working at 1.5 GHz and is equipped with 1-GB DDR3 RAM. The processor provides single issue and in-order microarchitecture with 8-stage pipeline. The ARM Cortex-A7 processor is working at 0.9 GHz and is equipped with 1-GB DDR3 RAM. The processor provides partial dual-issue and in-order microarchitecture with 8-stage pipeline. The ARM Cortex-A15 processor is working at 2 GHz and is equipped with 2-GB DDR3 RAM. The processor provides a 15-stage integer pipeline and a 17- to 25-stage floating point pipeline, with out-of-order speculative issue three-way super-scalar execution pipeline.

3.2 Modular Multiplication on 32-bit ARMv7 Cortex-A

The modular multiplication is the most expensive cryptography primitive in several Public Key Cryptography (i.e., RSA, ECC, SIDH, and SIKE). For this reason, previous works mainly focused on high-speed modular multiplication and squaring operations. Implementations are largely divided into two categories. One is the non-redundant representation (i.e., general implementation), and the other one is the redundant representation (i.e., specialized implementation).

Many works suggested high-speed implementations based on the non-redundant representation. In SAC'13, Bos et al. evaluated the performance of Montgomery multiplication on ARM–NEON processors [Bos et al. 2013]. They presented an approach to split the Montgomery multiplication into two parts, which can be computed in parallel. They flip the sign of the pre-computed Montgomery constant value and accumulate the result in two separate intermediate values that are computed. This approach can efficiently perform the Montgomery multiplication in the non-redundant representation. In Martins and Sousa [2014] and Martins and Sousa [2015], several

Montgomery multiplication methods were evaluated on 32-bit ARM Cortex-A15 processor. They concluded that the proposed Finely Integrated Operand Scanning achieved the fastest performance compared with Separated Operand Scanning. In WISA'14, Seo et al. introduced the COS method to speed up multi-precision multiplication on ARM–NEON architectures [Seo et al. 2014]. They developed the COS technique with the goal of reducing Read-After-Write (RAW) dependencies in the propagation of carries, which also reduces the number of pipeline stalls. The COS method operates on 32-bit words in a row-wise fashion and does not require redundant representation. In Seo et al. [2016], the Double Operand Scanning (DOS) method to speed-up multi-precision squaring with non-redundant representations on ARM-NEON architecture was presented. The DOS technique partly doubles the operands and computes the squaring operation without RAW dependencies between source and destination variables. In CHES'18, a unified ARM/NEON multi-precision multiplication for 32-bit ARMv7 Cortex-A processors was proposed [Seo et al. 2018]. The method finely integrated ARM and NEON instructions to exploit ARMâs instruction level parallelism. This approach reduces the number of memory accesses by employing both ARM and NEON registers for temporal storage and reduces the number of pipeline stalls even in processors with out-of-order execution capabilities, such as ARM Cortex-A15 processors.

Alternative implementations based on the redundant representation have been actively studied. The redundant representation (e.g., radix-28) does not fully utilize bits of word (e.g., 32-bit). Instead, remaining bits (e.g., 4-bit) are utilized for carry or borrow bit. This approach avoids carry or borrow propagation. In CHES 2012, Bernstein and Schwabe used the reduced radix ($2^{25.5}$) for an efficient modular multiplication of Curve25519 [Bernstein and Schwabe 2012]. They utilized the fast reduction to accelerate the performance of signature generation and verification. In HPEC 2013, a multiplicand reduction method in the reduced-radix representation was introduced for implementations of NIST P-192 and P-224 curves [Pabbuleti et al. 2013]. In particular, radix-24 and radix-28 were used for NIST P-192 and P-224 curves, respectively. In CHES 2014, the high-speed implementation of Curve41417 was proposed [Bernstein et al. 2014]. they utilized the reduced radix ($2^{25.875}$) and two-level Karatsuba multiplication. The modular multiplication combined refined Karatsuba and modular reduction to reduce the number of addition operations. In ICISC'15, the speed record of NIST P-521 over ARM-NEON platform in redundant representation ($2^{26.1}$) was proposed [Seo et al. 2015]. They exploited 1-level refined Karatsuba method to provide asymptotically faster integer multiplication and fast reduction algorithms. In SAC'16, they presented a high-speed and high-security implementation of the Four$\mathbb{Q}$ for 32-bit ARM Cortex-A processors with NEON support [Longa 2016]. They utilized the reduced radix (2^{26}) and achieved the faster performance than genus 2 Kummer and Curve25519 implementations by between 1.3× and 1.7× and between 2.1× and 2.4×, respectively. In conclusion, the redundant representation usually shows better performance than the non-redundant representation for the SIMD architecture when data or task parallelism is available.

In this article, we first implement the SIDH/SIKE protocol in the redundant representation on 32-bit ARMv7 Cortex-A processors. Unlike previous ECC implementations with redundant representation (i.e., Mersenne prime), SIDH/SIKE primes are inefficient with the fast reduction, while Montgomery reduction can be efficiently handled. We used radix-28 and refined Karatsuba method. The detailed comparison is given in Table 1.

4 OPTIMIZED SIDH/SIKE ARITHMETIC ON ARM–NEON

To perform SIDH/SIKE protocols, a number of primitive operations should be efficiently implemented. Among them, we focused on cryptographic primitive operations. In Figure 4, the overview of SIDH/SIKE computations are given. The bottom line is finite field arithmetic, which is primitive operations of SIDH/SIKE. By using finite field arithmetic, above operations (i.e., $\mathbb{F}_{p^2}$, group

Table 1. Comparison of Reduced Radix-based Implementations on 32-bit ARMv7

Implementation	Cryptography	Radix	Limb	Karatsuba	Reduction
Bernstein and Schwabe [2012]	Curve25519	$2^{25.5}$	10	–	Fast reduction
Pabbuleti et al. [2013]	NIST P-192	2^{24}	8	–	Fast reduction
Pabbuleti et al. [2013]	NIST P-224	2^{28}	8	–	Fast reduction
Bernstein et al. [2014]	Curve41417	$2^{25.875}$	16	√	Fast reduction
Seo et al. [2015]	NIST P-521	$2^{26.1}$	20	√	Fast reduction
Longa [2016]	Four$\mathbb{Q}$	2^{26}	5	–	Fast reduction
This work	SIKEp503	2^{28}	18	√	Montgomery reduction

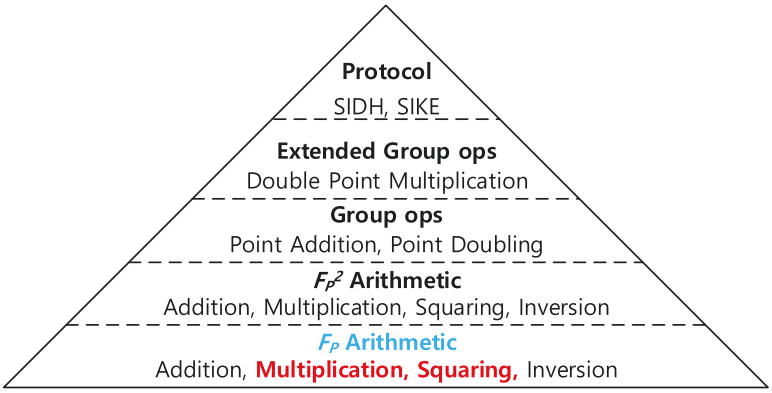

Fig. 4. Overview of SIDH/SIKE computations.

operation, extended group operation, and post-quantum protocol) are efficiently executed. Since the modular multiplication operation occupies the highest computation overheads among finite field arithmetic, we focused on optimizing the modular multiplication to improve the SIDH/SIKE protocol.

4.1 Multi-precision Multiplication

The prime of SIDHp503/SIKEp503 is $2^{250} \cdot 3^{159} - 1$ and 503-bit long. We chose radix-28 to align the operand and intermediate result. We increased a length of operand by 1-bit long to match the length of lower and higher parts of operand. The extended 504-bit operand is divided into 18-limb with radix-28 ($\frac{504}{28}$) as follows:

$$(28, 28, 28, 28, 28, 28, 28, 28, 28 \,||\, 28, 28, 28, 28, 28, 28, 28, 28, 28).$$

Since lower and higher 252-bitwise operands are properly aligned, we can directly apply Karatsuba multiplication, which replaces the one 504-bit multiplication complexity to three 252-bit multiplications with some addition and subtraction operations. In particular, we used the refined Karatsuba algorithm, which optimizes addition operations further [Bernstein 2009]. The implementation of Karatsuba algorithm is efficient with the NEON engine. The NEON engine supports SIMD instructions, which can process multiple data in single instruction. Furthermore, the NEON engine has large space of registers to maintain the values. The NEON engine over 32-bit ARMv7 Cortex-A provides sixteen 128-bitwise registers.[3] The sub-routines of 504-bit Karatsuba multiplication are

[3]64-bit ARMv8 Cortex-A provides 32 128-bitwise registers.

performing three 252-bit multiplication operations. For the 252-bit multiplication, we need 5 registers for both 18-limb of 32-bit operands and 9 registers for 18-limb of 64-bit intermediate results and 1 register for temporal storage.

For the memory efficiency, whole results are not maintained. Intermediate results are accumulated and only part of results are stored to the memory. By implementing multiplication and modular reduction operations in an integrated form, the memory access is optimized. Detailed descriptions are given in Algorithm 1.

ALGORITHM 1: Refined Karatsuba-based multiplication for 504-bit

Require: Integer a, b satisfying $1 \leq log_2 a, log_2 b \leq 504$.
Ensure: Results $c = a \cdot b$ in memory m (m_3, m_2, m_1, m_0).

1: $a_L \leftarrow a \; mod \; 2^{252}$
2: $a_H \leftarrow a \; div \; 2^{252}$
3: $b_L \leftarrow b \; mod \; 2^{252}$
4: $b_H \leftarrow b \; div \; 2^{252}$
5: $c_L \leftarrow a_L \cdot b_L$ {lower 252-bit multiplication}
6: $m_0 \leftarrow c_L \; mod \; 2^{252}$ {saving lower part to memory}
7: $t \leftarrow c_L \; div \; 2^{252} - (a_H \cdot b_H \; mod \; 2^{252}) + (a_H \cdot b_H \; div \; 2^{252}) \cdot 2^{252}$ {higher 252-bit multiplication}
8: $m_3 \leftarrow t \; div \; 2^{252}$ {saving higher part to memory}
9: $t \leftarrow t + (t \; mod \; 2^{252}) \cdot 2^{252}$
10: $t \leftarrow (t \; mod \; 2^{252} - m_0) - (t \; div \; 2^{252}) \cdot 2^{252}$
11: $a_K \leftarrow a_L + a_H$
12: $b_K \leftarrow b_L + b_H$
13: $c_K \leftarrow t + a_K \cdot b_K$ {middle 252-bit multiplication}
14: $(m_2, m_1) \leftarrow c_K$ {saving middle part to memory; this can be optimized away}

In the beginning, both operands are divided into lower and higher parts from Steps 1–4. Both lower and higher parts are 252-bit long. In Step 5, the lower part of operands (a_L, b_L) are multiplied and output the result (c_L). In Step 6, the lower part of result (c_L) is stored into memory (m_0; i.e., stack or heap) and flushed from registers. In Step 7, the higher part of intermediate result $(c_L \; div \; 2^{252})$ is maintained in NEON registers to accumulate the result. When the higher part of operands (a_H, b_H) are multiplied, the lower part $(a_H \cdot b_H \; mod \; 2^{252})$ and higher part $(a_H \cdot b_H \; div \; 2^{252})$ of intermediate results are directly subtracted and added to the result $(c_L \; div \; 2^{252})$, respectively. These operations are efficiently handled with `VMLSL.S32` and `VMLAL.S32` instructions, which ensure multiplication together with subtraction/addition operations. In Step 8, the higher part of result $(t \; div \; 2^{252})$ is stored into memory (m_3; i.e., stack or heap). In Step 9, the lower part of intermediate result $(t \; mod \; 2^{252})$ is added to the higher part of intermediate result. In Step 10, the lower part of intermediate result is subtracted by result in m_0 $(a_L \cdot b_L \bmod 2^{252})$ and the higher part of intermediate result $(t \; div \; 2^{252})$ is negated. In Steps 11 and 12, higher and lower parts of operands are added each other. In Step 13, operands (a_K and b_K) are multiplied and added to the intermediate result (t). Finally, results (c_K) are stored in memory (m_2, m_1).

For the 252-bit multiplication, we perform 9-limb multiplication in radix-28 representation (i.e., Step 5, 7, and 13 of Algorithm 1). Notations ($a_0 \sim a_8$ and $b_0 \sim b_8$) indicate both operands in 28-bit long and other notations ($c_0 \sim c_{16}$) represent intermediate results in 56-bit or above. Detailed

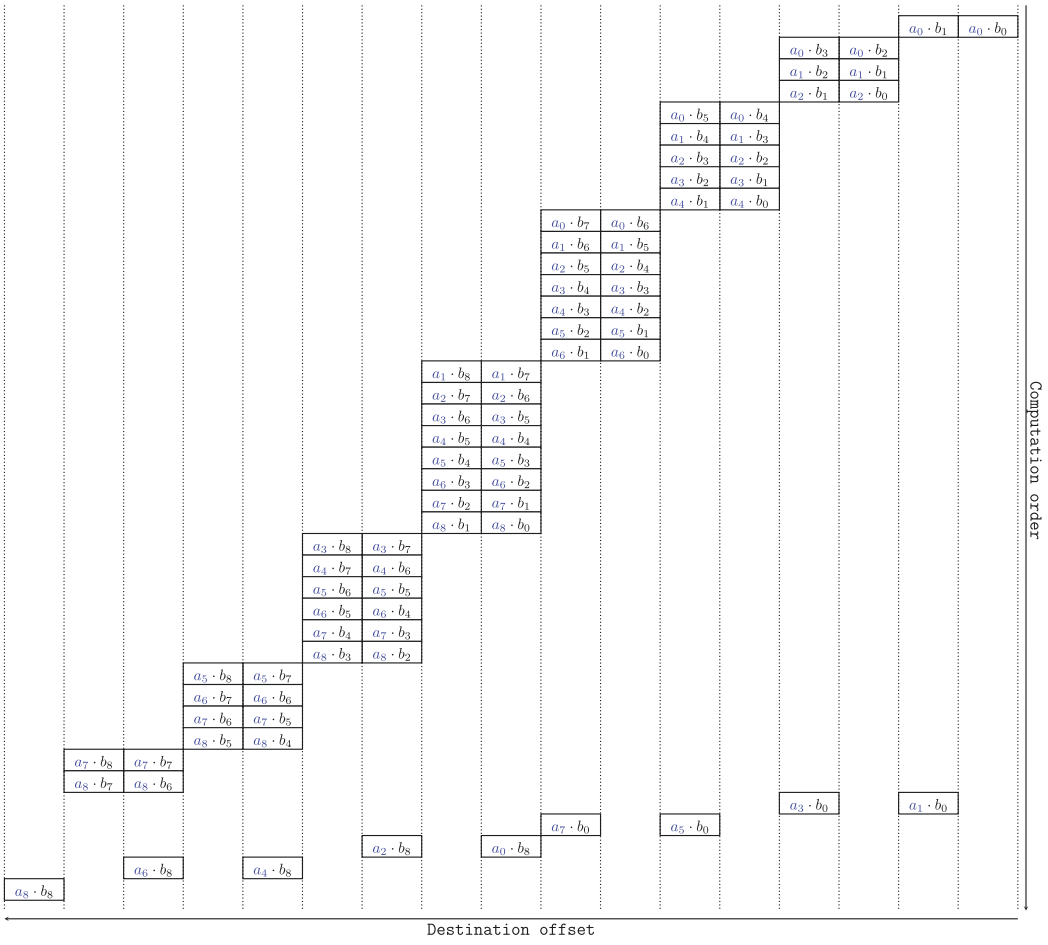

Fig. 5. Illustration of 252-bit multiplication in 2-way data parallelism.

descriptions of 252-bit multiplication on 28-radix is as follows:

$$
\begin{aligned}
c_0 &\leftarrow a_0b_0\\
c_1 &\leftarrow a_0b_1 + a_1b_0\\
c_2 &\leftarrow a_0b_2 + a_2b_0 + a_1b_1\\
c_3 &\leftarrow a_0b_3 + a_3b_0 + a_1b_2 + a_2b_1\\
c_4 &\leftarrow a_0b_4 + a_4b_0 + a_1b_3 + a_3b_1 + a_2b_2\\
c_5 &\leftarrow a_0b_5 + a_5b_0 + a_1b_4 + a_4b_1 + a_2b_3 + a_3b_2\\
c_6 &\leftarrow a_0b_6 + a_6b_0 + a_1b_5 + a_5b_1 + a_2b_4 + a_4b_2 + a_3b_3\\
c_7 &\leftarrow a_0b_7 + a_7b_0 + a_1b_6 + a_6b_1 + a_2b_5 + a_5b_2 + a_3b_4 + a_4b_3\\
c_8 &\leftarrow a_0b_8 + a_8b_0 + a_1b_7 + a_7b_1 + a_2b_6 + a_6b_2 + a_3b_5 + a_5b_3 + a_4b_4\\
c_9 &\leftarrow a_1b_8 + a_8b_1 + a_2b_7 + a_7b_2 + a_3b_6 + a_6b_3 + a_4b_5 + a_5b_4\\
c_{10} &\leftarrow a_2b_8 + a_8b_2 + a_3b_7 + a_7b_3 + a_4b_6 + a_6b_4 + a_5b_5\\
c_{11} &\leftarrow a_3b_8 + a_8b_3 + a_4b_7 + a_7b_4 + a_5b_6 + a_6b_5\\
c_{12} &\leftarrow a_4b_8 + a_8b_4 + a_5b_7 + a_7b_5 + a_6b_6\\
c_{13} &\leftarrow a_5b_8 + a_8b_5 + a_6b_7 + a_7b_6\\
c_{14} &\leftarrow a_6b_8 + a_8b_6 + a_7b_7\\
c_{15} &\leftarrow a_7b_8 + a_8b_7\\
c_{16} &\leftarrow a_8b_8
\end{aligned}
$$

There are two types of multiplication instructions, including UMULL and UMLAL instructions. When we need to initialize the register, we used UMULL instruction. For multiplication and accumulation step, the UMLAL instruction is more efficient than general UMULL instruction.

For the data parallel computation, the above multiplication routine should be re-written in a SIMD friendly form. Particularly, the NEON architecture supports 2-way 32-bitwise multiplication, which means two concurrent 32-bit multiplications are computed and results are stored in two consecutive 64-bit results (i.e., 128-bit register). For this reason, alignments in the 128-bit register should be concerned to accumulate multiplication results into correct columns. We group two adjacent partial products as follows: $(c_1, c_0), (c_3, c_2), (c_5, c_4), (c_7, c_6), (c_9, c_8), (c_{11}, c_{10}), (c_{13}, c_{12}), (c_{15}, c_{14})$. However, all partial products cannot be grouped in original operand (b) alignment. We re-located the operand by conducting the shift to left by word size. The original operand alignment of b is $(b_1, b_0), (b_3, b_2), (b_5, b_4), (b_7, b_6), b_8$. The shifted operand is $(b_2, b_1), (b_4, b_3), (b_6, b_5), (b_8, b_7)$. This can be done with the VEXT instruction and detailed descriptions are as follows:

```
  ⋮
VEXT.S32 Q3, Q3, Q4     // Q3 ← (b2, b1, b4, b3)
VEXT.S32 Q4, Q4, Q5     // Q4 ← (b6, b5, b8, b7)
  ⋮
```

Afterward, vectorized partial products can be performed. Still, some partial products $(a_7b_0, a_5b_0, a_3b_0, a_1b_0, a_6b_8, a_4b_8, a_2b_8, a_0b_8)$ are not properly aligned in groupwise. Results are first performed in groupwise and the result is aligned, sequentially. The complete 252-bit multiplication in two-way data parallelism, as illustrated in Figure 5, is as follows:

$$
\begin{aligned}
(c_1, c_0) &\leftarrow (a_0b_1, a_0b_0)\\
(c_3, c_2) &\leftarrow (a_0b_3, a_0b_2) + (a_1b_2, a_1b_1) + (a_2b_1, a_2b_0)\\
(c_5, c_4) &\leftarrow (a_0b_5, a_0b_4) + (a_1b_4, a_1b_3) + (a_2b_3, a_2b_2) + (a_3b_2, a_3b_1)\\
&\quad + (a_4b_1, a_4b_0)\\
(c_7, c_6) &\leftarrow (a_0b_7, a_0b_6) + (a_1b_6, a_1b_5) + (a_2b_5, a_2b_4) + (a_3b_4, a_3b_3)\\
&\quad + (a_4b_3, a_4b_2) + (a_5b_2, a_5b_1) + (a_6b_1, a_6b_0)\\
(c_9, c_8) &\leftarrow (a_1b_8, a_1b_7) + (a_2b_7, a_2b_6) + (a_3b_6, a_3b_5) + (a_4b_5, a_4b_4)\\
&\quad + (a_5b_4, a_5b_3) + (a_6b_3, a_6b_2) + (a_7b_2, a_7b_1) + (a_8b_1, a_8b_0)\\
(c_{11}, c_{10}) &\leftarrow (a_3b_8, a_3b_7) + (a_4b_7, a_4b_6) + (a_5b_6, a_5b_5) + (a_6b_5, a_6b_4)\\
&\quad + (a_7b_4, a_7b_3) + (a_8b_3, a_8b_2)\\
(c_{13}, c_{12}) &\leftarrow (a_5b_8, a_5b_7) + (a_6b_7, a_6b_6) + (a_7b_6, a_7b_5) + (a_8b_5, a_8b_4)\\
(c_{15}, c_{14}) &\leftarrow (a_7b_8, a_7b_7) + (a_8b_7, a_8b_6)\\
(t_1, t_0) &\leftarrow (a_3b_0, a_1b_0);\ c_1 \leftarrow c_1 + t_0;\ c_3 \leftarrow c_3 + t_1\\
(t_1, t_0) &\leftarrow (a_7b_0, a_5b_0);\ c_5 \leftarrow c_5 + t_0;\ c_7 \leftarrow c_7 + t_1\\
(t_1, t_0) &\leftarrow (a_2b_8, a_0b_8);\ c_8 \leftarrow c_8 + t_0;\ c_{10} \leftarrow c_{10} + t_1\\
(t_1, t_0) &\leftarrow (a_6b_8, a_4b_8);\ c_{12} \leftarrow c_{12} + t_0;\ c_{14} \leftarrow c_{14} + t_1\\
c_{16} &\leftarrow a_8b_8
\end{aligned}
$$

To avoid pipeline stall, partial products are updated to destination variables without RAW dependencies between source and destination variables. The order of computation follows operand-scanning method and detailed instructions are as follows:

```
  ⋮
VMULL.S32 Q5, D6, D0[0]   // (c1, c0) ← (a0b1, a0b0)
VMULL.S32 Q6, D7, D0[0]   // (c3, c2) ← (a0b3, a0b2)
VMULL.S32 Q7, D8, D0[0]   // (c5, c4) ← (a0b5, a0b4)
VMULL.S32 Q8, D9, D0[0]   // (c7, c6) ← (a0b7, a0b6)
  ⋮
VMLAL.S32 Q6, D6, D1[0]   // (c3, c2) ← (c3, c2) + (a2b1, a2b0)
VMLAL.S32 Q7, D7, D1[0]   // (c5, c4) ← (c5, c4) + (a2b3, a2b2)
VMLAL.S32 Q8, D8, D1[0]   // (c7, c6) ← (c7, c6) + (a2b5, a2b4)
VMULL.S32 Q9, D9, D1[0]   // (c9, c8) ← (a2b7, a2b6)
  ⋮
```

In the beginning, partial products ($a_0b_0 \sim b_7$) are computed and results are saved into registers (`Q5, Q6, Q7, Q8`) in a sequential order. Afterward, partial products with the operand a_2 is performed. At this time, we ensure that previous destination variables are not directly accessed in the next instruction, which incurs pipeline stalls. This kind of pipeline stall avoidance mechanism is working even for processors with out-of-order feature (i.e., 32-bit ARM Cortex-A15) [Seo et al. 2018].

4.2 Multi-precision Squaring

Multi-precision squaring can be implemented with ordinary multiplication methods. However, the squaring method has two distinguished features over multiplication methods. Both partial products ($a[i] \times a[j]$ and $a[j] \times a[i]$) output identical results. By taking account of this feature, parts can be calculated in multiplied with doubled form (i.e., $2 \times a[i] \times a[j]$) which provides same results of conventional multiplication (i.e., $a[i] \times a[j] + a[j] \times a[i]$) replacing one multiplication into one addition. We applied the squaring method to 252-bitwise operand. Unlike the multiplication operation, the squaring can eliminate the almost half of partial products with doubling technique. To perform doubling in an efficient way, double multiplication instructions, such as `VQDMULL`, `VQDMLAL`, and `VQDMLSL`, are utilized. These operations save one addition for each multiplication. Detailed squaring operations are as follows:

$$
\begin{aligned}
c_0 &\leftarrow a_0a_0 \\
c_1 &\leftarrow 2(a_0a_1) \\
c_2 &\leftarrow 2(a_0a_2) + a_1a_1 \\
c_3 &\leftarrow 2(a_0a_3 + a_1a_2) \\
c_4 &\leftarrow 2(a_0a_4 + a_1a_3) + a_2a_2 \\
c_5 &\leftarrow 2(a_0a_5 + a_1a_4 + a_2a_3) \\
c_6 &\leftarrow 2(a_0a_6 + a_1a_5 + a_2a_4) + a_3a_3 \\
c_7 &\leftarrow 2(a_0a_7 + a_1a_6 + a_2a_5 + a_3a_4) \\
c_8 &\leftarrow 2(a_0a_8 + a_1a_7 + a_2a_6 + a_3a_5) + a_4a_4 \\
c_9 &\leftarrow 2(a_1a_8 + a_2a_7 + a_3a_6 + a_4a_5) \\
c_{10} &\leftarrow 2(a_2a_8 + a_3a_7 + a_4a_6) + a_5a_5 \\
c_{11} &\leftarrow 2(a_3a_8 + a_4a_7 + a_5a_6) \\
c_{12} &\leftarrow 2(a_4a_8 + a_5a_7) + a_6a_6 \\
c_{13} &\leftarrow 2(a_5a_8 + a_6a_7) \\
c_{14} &\leftarrow 2(a_6a_8) + a_7a_7 \\
c_{15} &\leftarrow 2(a_7a_8) \\
c_{16} &\leftarrow a_8a_8
\end{aligned}
$$

Similarly to the SIMD friendly multiplication, the squaring operation also needs to group two intermediate results in SIMD friendly way. Detailed descriptions are as follows:

$$
\begin{aligned}
(c_1, c_0) &\leftarrow (a_0a_1, a_0a_0);\ c_1 \leftarrow 2c_1\\
(c_3, c_2) &\leftarrow (a_1a_2, a_1a_1);\ c_3 \leftarrow 2c_3\\
(c_3, c_2) &\leftarrow (c_3, c_2) + 2(a_0a_3, a_0a_2)\\
(c_5, c_4) &\leftarrow (a_2a_3, a_2a_2);\ c_5 \leftarrow 2c_5\\
(c_5, c_4) &\leftarrow (c_5, c_4) + 2(a_0a_5, a_0a_4) + 2(a_1a_4, a_1a_3)\\
(c_7, c_6) &\leftarrow (a_3a_4, a_3a_3);\ c_7 \leftarrow 2c_7\\
(c_7, c_6) &\leftarrow (c_7, c_6) + 2(a_0a_7, a_0a_6) + 2(a_1a_6, a_1a_5) + 2(a_2a_5, a_2a_4)\\
(c_9, c_8) &\leftarrow (a_5a_5, a_4a_4);\ (c_{11}, c_{10}) \leftarrow (0, 0);\ c_{10} \leftrightarrow c_9\\
(c_{13}, c_{12}) &\leftarrow (a_7a_7, a_6a_6);\ (c_{15}, c_{14}) \leftarrow (0, 0);\ c_{14} \leftrightarrow c_{13}\\
c_{16} &\leftarrow a_8a_8\\
(c_9, c_8) &\leftarrow (c_9, c_8) + 2(a_5a_4, a_5a_3) + 2(a_6a_3, a_6a_2) + 2(a_7a_2, a_7a_1)\\
&\quad + 2(a_8a_1, a_8a_0)\\
(c_{11}, c_{10}) &\leftarrow (c_{11}, c_{10}) + 2(a_6a_5, a_6a_4) + 2(a_7a_4, a_7a_3) + 2(a_8a_3, a_8a_2)\\
(c_{13}, c_{12}) &\leftarrow (c_{13}, c_{12}) + 2(a_7a_6, a_7a_5) + 2(a_8a_5, a_8a_4)\\
(c_{15}, c_{14}) &\leftarrow (c_{15}, c_{14}) + 2(a_8a_7, a_8a_6)
\end{aligned}
$$

As we noted that half of partial products are optimized with simple doubling. Some grouped partial products consists of normal product and doubled product. In this case, grouped partial product is performed first and one product is doubled with addition instruction (`VADD.S64`). Four partial products $(a_4a_4, a_5a_5, a_6a_6, a_7a_7)$ are performed in a groupwise but the destination is not aligned properly. These four products are performed in a groupwise and results are swapped with initialized value for efficient result accumulation. Detailed descriptions are as follows:

```
  ⋮
VMULL.S32 Q9, D2, D2      // (c9, c8) ← (a5a5, a4a4)
VEOR      Q10, Q10, Q10   // Initialization of (c11, c10)
VSWP      D19, D20        // c10 ↔ c9
  ⋮
```

4.3 Modular Reduction

Modular reduction is a performance-critical building block of SIDH/SIKE based post-quantum cryptography. One of the most well-known techniques used for its implementation is Montgomery reduction [Montgomery 1985]. A basic description of Montgomery reduction is depicted in Algorithm 2.

ALGORITHM 2: Montgomery reduction

Require: An odd modulus m, the Montgomery radix $r > m$, an operand $c \in [0, m^2 - 1]$, and the precomputed constant $m' = -m^{-1} \bmod r$

Ensure: Montgomery product $z = \text{MonRed}(c, r) = c \cdot r^{-1} \bmod m$

1: $q \leftarrow c \cdot m' \bmod r$
2: $z \leftarrow (c + q \cdot m)/r$
3: **if** $z \geq m$ **then** $z \leftarrow z - m$
4: **return** z

The efficient implementation of Montgomery multiplication has been actively studied for ARM architectures. However, previous SIDH implementation focused on the non-redundant representation for Montgomery reduction.

First SIDH implementation on 32-bit ARMv7 Cortex-A used the COS method, issuing two multiplications at once and finely re-ordering the computation routines to avoid pipeline stalls

[Seo et al. 2014, 2016; Koziel et al. 2016; Jalali et al. 2018]. However, the non-redundant representation consumes long execution timing for carry propagation routine.

In CHES'18, Seo et al. suggested the variant of Hybrid-Scanning (HS) for "SIDH-friendly" Montgomery reduction on 32-bit ARMv7 Cortex-A processors [Seo et al. 2018]. The basic structure follows Operand-Scanning (OS) method but the whole computation is divided into two parts. Both parts are computed in parallel way by exploiting both ARM and NEON Arithmetic Logic Unit. This work still utilized the non-redundant representation to ensure compatibility with Microsoft SIDH library.

In this article, we firstly applied the redundant representation for "SIDH-friendly" Montgomery reduction. Montgomery multiplication for SIDH/SIKE can be simplified by taking advantage of so-called "Montgomery-friendly" modulus, which admits efficient computations, such as *all-zero* words for lower part of the modulus. These all-zero words can be ignored during computation.

Efficient optimizations for the modulus were first pointed out by Costello et al. [2016] in the setting of SIDH when using modulus of the form $2^x \cdot 3^y - 1$ (referred to as "SIDH-friendly" primes) are exploited by the SIDH library [Costello et al. 2018]. The simplified equation for SIDHp503, where $r = 2^{504}$, is given as follows:

$$\begin{aligned} z &= (c + (c \cdot m' \bmod 2^{504}) \cdot m)/2^{504} \\ &= (c + (c \cdot m' \bmod 2^{504}) \cdot (2^{250} \cdot 3^{159}) - (c \cdot m' \bmod 2^{504}))/2^{504} \\ &= (c + (c \cdot m' \bmod 2^{504}) \cdot (2^{250} \cdot 3^{159}))/2^{504} \end{aligned}$$

The equation $((c \cdot m' \bmod 2^{504}))/2^{504})$ can be optimized away. For this reason, the modulus with one addition is efficiently performing the Montgomery reduction and radix-28 representation is as follows:

$$\begin{aligned} m + 1 := &\texttt{4066F54 1811E1E 6045C6B DDA77A4 D01B9BF 6C87B7E,} \\ &\texttt{7DAF130 85BDA22 11E7A0A C000000 0000000 0000000,} \\ &\texttt{0000000 0000000 0000000 0000000 0000000 0000000} \end{aligned}$$

As we can see the modulus with one addition, the lower 8-limb is filled with *all-zero* bits, which can be optimized away. Only higher 10-limb is used for the Montgomery reduction.

In Step 14 of Algorithm 1, the result (c_K) is stored into memory (m_2, m_1). For the efficient implementation, Montgomery reduction is directly performed after multiplication. In this case, the result (c_K) is kept in NEON registers and Montgomery reduction is performed while accumulating the output on the intermediate result (c_K), which optimizes the 504-bit memory load and store.

Montgomery reduction consists of similar computations of multiplication. Only two features are different. First, partial products are accumulated to the intermediate result of multi-precision multiplication. This can be performed with `VMLAL.S32` instruction. Second, the length of operand is asymmetric. For the case of SIDHp503/SIKEp503, the number of limb for modulus and quotient are 10 (280-bit) and 18 (504-bit), respectively. We divide the multiplication of modulus and quotient into 2 blocks since the number of NEON register is limited to maintain values. By calculating required number of registers, the optimal block size is around 9- to 10-limb in NEON. Detailed descriptions are given in Algorithm 3. The Algorithm consists of three steps including first block computation, second block computation, and finalization. In Steps 1–6, quotients (q) are obtained from intermediate results (c). Since the multiplication is performed in radix-28, the quotient should be properly adjusted to radix-28. In Steps 7–12, the multiplication of modulus and quotient ($m \cdot q$) are accumulated to the intermediate result (c). From Step 13, second block is performed. Similarly, in Steps 13–18, quotients (q) are obtained from intermediate results (c). Afterward, the remaining multiplication of modulus and quotient is performed. Before generating the final result, the final reduction and radix adjustment are performed. Detailed descriptions of final reduction and radix adjustment are given in Section 4.4 and 4.5, respectively.

ALGORITHM 3: Blockwise scanning for Montgomery reduction of SIDHp503 in 28-radix

Require: Intermediate result c ($c_{17} \sim c_0$) $= a \cdot b$ where $c \in [0, m^2 - 1]$, modulus $m + 1$ ($m_{17} \sim m_8$).
Ensure: Montgomery product $z = \text{MonRed}(c, r) = t \cdot r^{-1} \bmod m$

First block computation
1: **for** $i = 0$ to 7 by 1 **do**
2: $\quad q_i = c_i \bmod 2^{28}$
3: $\quad c_{i+1} = c_{i+1} + (c_i \gg 28)$
4: $c_8 = c_8 + m_8 \cdot q_0$
5: $q_8 = c_8 \bmod 2^{28}$
6: $c_9 = c_9 + (c_8 \gg 28)$

7: **for** $i = 9$ to 16 by 1 **do**
8: $\quad$ **for** $j = 7$ to i by 1 **do**
9: $\quad\quad c_i = c_i + m_{j+1} \cdot q_{i-j-1}$
10: **for** $i = 17$ to 25 by 1 **do**
11: $\quad$ **for** $j = 26$ to i by -1 **do**
12: $\quad\quad c_i = c_i + m_{35-j} \cdot q_{j-i-1}$

Second block computation
13: **for** $i = 9$ to 16 by 1 **do**
14: $\quad q_i = c_i \bmod 2^{28}$
15: $\quad c_{i+1} = c_{i+1} + (c_i \gg 28)$
16: $c_{17} = c_{17} + m_8 \cdot q_9$
17: $q_{17} = c_{17} \bmod 2^{28}$
18: $c_{18} = c_{18} + (c_{17} \gg 28)$

19: **for** $i = 18$ to 25 by 1 **do**
20: $\quad$ **for** $j = 7$ to $i - 9$ by 1 **do**
21: $\quad\quad c_i = c_i + m_{j+1} \cdot q_{i-j-1}$
22: **for** $i = 26$ to 34 by 1 **do**
23: $\quad$ **for** $j = 26$ to $i - 9$ by -1 **do**
24: $\quad\quad c_i = c_i + m_{35-j} \cdot q_{j-i+17}$

Finalization
25: v = **final reduction** (c)
26: z = **radix adjustment** (v)

27: **return** z

The Montgomery reduction consists of a number of multiplication and accumulation routines. This routine can be efficiently handled with `VMLAL.S32` instruction. Similarly to multi-precision multiplication and squaring operations in two-way implementation, Montgomery reduction also needs to group two intermediate results in SIMD friendly way. The following is SIMD friendly Montgomery reduction for 1 block computation:

$$
\begin{aligned}
t_0 \leftarrow q_0 m_8;\ c_8 \leftarrow c_8 + t_0;\ q_8 \leftarrow c_8 \&\texttt{0xFFFFFFF};\ c_9 \leftarrow c_9 + c_8 \gg 28 \\
(c_{10}, c_9) \leftarrow (c_{10}, c_9) + (q_0 m_{10}, q_0 m_9) + (q_1 m_9, q_1 m_8) \\
(c_{12}, c_{11}) \leftarrow (c_{12}, c_{11}) + (q_0 m_{12}, q_0 m_{11}) + (q_1 m_{11}, q_1 m_{10}) + (q_2 m_{10}, q_2 m_9) \\
+ (q_3 m_9, q_3 m_8) \\
(c_{14}, c_{13}) \leftarrow (c_{14}, c_{13}) + (q_0 m_{14}, q_0 m_{13}) + (q_1 m_{13}, q_1 m_{12}) \\
+ (q_2 m_{12}, q_2 m_{11}) + (q_3 m_{11}, q_3 m_{10}) + (q_4 m_{10}, q_4 m_9) + (q_5 m_9, q_5 m_8) \\
(c_{16}, c_{15}) \leftarrow (c_{16}, c_{15}) + (q_0 m_{16}, q_0 m_{15}) + (q_1 m_{15}, q_1 m_{14}) + (q_2 m_{14}, q_2 m_{13}) \\
+ (q_3 m_{13}, q_3 m_{12}) + (q_4 m_{12}, q_4 m_{11}) + (q_5 m_{11}, q_5 m_{10}) + (q_6 m_{10}, q_6 m_9) \\
+ (q_7 m_9, q_7 m_8) \\
(c_{18}, c_{17}) \leftarrow (c_{18}, c_{17}) + (q_1 m_{17}, q_1 m_{16}) + (q_2 m_{16}, q_2 m_{15}) + (q_3 m_{15}, q_3 m_{14}) \\
+ (q_4 m_{14}, q_4 m_{13}) + (q_5 m_{13}, q_5 m_{12}) + (q_6 m_{12}, q_6 m_{11}) + (q_7 m_{11}, q_7 m_{10}) \\
+ (q_8 m_{10}, q_8 m_9) \\
(c_{20}, c_{19}) \leftarrow (c_{20}, c_{19}) + (q_3 m_{17}, q_3 m_{16}) + (q_4 m_{16}, q_4 m_{15}) + (q_5 m_{15}, q_5 m_{14}) \\
+ (q_6 m_{14}, q_6 m_{13}) + (q_7 m_{13}, q_7 m_{12}) + (q_8 m_{12}, q_8 m_{11}) \\
(c_{22}, c_{21}) \leftarrow (c_{22}, c_{21}) + (q_5 m_{17}, q_5 m_{16}) + (q_6 m_{16}, q_6 m_{15}) + (q_7 m_{15}, q_7 m_{14}) \\
+ (q_8 m_{14}, q_8 m_{13})(c_{24}, c_{23}) \leftarrow (c_{24}, c_{23}) + (q_7 m_{17}, q_7 m_{16}) + (q_8 m_{16}, q_8 m_{15}) \\
t_0 \leftarrow q_2 m_8;\ c_{10} \leftarrow c_{10} + t_0 \\
(t_1, t_0) \leftarrow (q_6 m_8, q_4 m_8);\ c_{12} \leftarrow c_{12} + t_0;\ c_{14} \leftarrow c_{14} + t_1 \\
t_0 \leftarrow q_8 m_8;\ c_{16} \leftarrow c_{16} + t_0 \\
(t_1, t_0) \leftarrow (q_2 m_{17}, q_0 m_{17});\ c_{17} \leftarrow c_{17} + t_0;\ c_{19} \leftarrow c_{19} + t_1 \\
(t_1, t_0) \leftarrow (q_6 m_{17}, q_4 m_{17});\ c_{21} \leftarrow c_{21} + t_0;\ c_{23} \leftarrow c_{23} + t_1 \\
t_0 \leftarrow q_8 m_{17};\ c_{25} \leftarrow c_{25} + t_0
\end{aligned}
$$

In the beginning, the quotient (q_8) is extracted from the result (c_8) and upper part of c_8 is added to the c_9. Afterward, two-way multiplication routines are performed. Among them, six partial products (q_6m_8, q_4m_8, q_6m_{17}, q_4m_{17}, q_2m_{17}, q_0m_{17}) can be performed in groupwise but the destination is not aligned properly. For this reason, these six products are performed in groupwise and results are swapped with initialized value for efficient result accumulation. The other partial products ($q_2m_8, q_8m_8, q_8m_{17}$) are performed in single multiplication and added to the result.

4.4 Final Reduction

The last step of Montgomery reduction is final reduction. This reduces the result once again with simple subtraction. Unlike the redundant implementation for Mersenne prime, the modulus of SIDHp503 ($2^{250} \cdot 3^{159} - 1$) cannot take advantages of fast reduction. For the SIDHp503 case, we perform the subtraction by observing carry bits. In Algorithm 4, the optimized final reduction for SIDHp503 is given.

ALGORITHM 4: Final reduction for SIDHp503

Require: Result $c \in [0, 504]$.
Ensure: Reduced result $v = c - (m \cdot c \gg 503)$.
1: $carry \leftarrow c \gg 503$
2: $c \leftarrow c - carry \cdot (m + 1)$
3: $v \leftarrow c + carry$
4: **return** v

First, carry bits are extracted from the result (c) over 503-bit. Afterward, the result (c) is subtracted by the partial product with carry bits ($carry$) and modulus ($m + 1$). Straightforward multiplication requires 18 partial products (9 groupwise products). However, the proposed final reduction replaces 18 partial products to 10 partial products (5 groupwise products) and 1 addition since the multiplication of carry bits and SIDHp503 prime can be rewritten as follows:

$$-carry \cdot m = -carry \cdot (m + 1) + carry.$$

The left and right parts of above equation generate the identical value. The proposed final reduction is implemented in following instructions:

```
  ⋮
VSHR.S64  D0,D27,#27 // carry (CA) extraction
VADD.S64  D10,D10,D0 // c0 ← c0 + CA
VMLSL.S32 Q9,D4,D0[0] // (c9, c8) ← (c9, c8) − (m9, m8) · CA
VMLSL.S32 Q10,D5,D0[0] // (c11, c10) ← (c11, c10) − (m11, m10) · CA
VMLSL.S32 Q11,D6,D0[0] // (c13, c12) ← (c13, c12) − (m13, m12) · CA
VMLSL.S32 Q12,D7,D0[0] // (c15, c14) ← (c15, c14) − (m15, m14) · CA
VMLSL.S32 Q13,D8,D0[0] // (c17, c16) ← (c17, c16) − (m17, m16) · CA
  ⋮
```

Carry bits (`D0`) are extracted with `VSHR.S64` instruction and carry bits (`D0`) are added to c_0. Afterward, carry bits (`D0`) are multiplied by modulus (`D4` ∼ `D8`) and subtracted to the intermediate result (`Q9` ∼ `Q13`) with the `VMLSL` instruction.

During the computation, the result is not fully reduced to take an advantage of lazy reduction. The full reduction is performed at the last step of SIDH protocols.

Table 2. Comparison of Modular Multiplication Method for SIDH Implementations on 32-bit ARMv7 Cortex-A Processors

Method	Representation	Instruction	Technique	Karatsuba
Koziel et al. [2016]	non-redundant	NEON	COS	√
Jalali et al. [2018]	non-redundant	NEON	COS	√
Seo et al. [2018]	non-redundant	ARM/NEON	HS	√
This work	redundant	NEON	OS	√

4.5 Radix Adjustments

Multiplication and squaring computations produce a product of 63-bit 18 limbs for intermediate results. These values cannot be used in following computations due to overflow or underflow. For this reason, we use a sequence of carries to bring each limb down to 26 or 27 bits. We vectorized between a carry $c_0 \rightarrow c_1$ and $c_9 \rightarrow c_{10}$, between a carry $c_1 \rightarrow c_2$ and $c_{10} \rightarrow c_{11}$. The computation order is as follows: $(c_9, c_0) \rightarrow (c_{10}, c_1)$, $(c_{10}, c_1) \rightarrow (c_{11}, c_2)$, $(c_{11}, c_2) \rightarrow (c_{12}, c_3)$, $(c_{12}, c_3) \rightarrow (c_{13}, c_4)$, $(c_{13}, c_4) \rightarrow (c_{14}, c_5)$, $(c_{14}, c_5) \rightarrow (c_{15}, c_6)$, $(c_{15}, c_6) \rightarrow (c_{16}, c_7)$, $(c_{16}, c_7) \rightarrow (c_{17}, c_8)$. The computations output 18 limbs of results (28, 28, 28, 28, 28, 28, 28, 28, 29 || 28, 28, 28, 28, 28, 28, 28, 28, 29). The highest limb is set to radix-29 and the others are set to radix-28.

4.6 Modular Addition and Subtraction

The NEON instruction supports 4-way 32-bitwise addition (`VADD.S32`) or subtraction (`VSUB.S32`). We utilized these instructions to perform addition or subtraction. Afterward, final reduction is performed with Algorithm 4. In this case, `VMLS.I32` instruction is utilized, which ensures four-way 32-bitwise multiplication and subtraction, since the result of addition or subtraction is between 28- and 30-bit.

5 PERFORMANCE EVALUATION

In this section, we evaluate the performance of proposed algorithms for 32-bit ARMv7 Cortex-A processors. All our finite field implementations were written in assembly language and higher layer (group operation and protocol) is written in C language. The program is compiled with GCC 5.4.0 in an optimization level `-O3`[4]. The operating system is Ubuntu 16.04. Target processors are 32-bit ARM Cortex-A5, A7, and A15.

To evaluate the performance of the proposed Montgomery multiplication specialized to the setting of SIDH/SIKE, we integrate it into the Microsoft SIDH/SIKE library [Costello et al. 2018]. This library includes the SIDH/SIKE protocols using the parameters sets SIDHp503/SIKEp503 based on the 503-bit [Costello et al. 2016; Azarderakhsh et al. 2019]. In this article, we only include the SIDHp503/SIKEp503 implementation as a proof-of-concept but other parameters (i.e., SIDHp434/SIKEp434, SIDHp610/SIKEp610, and SIDHp751/SIKEp751) can be accelerated with proposed method without difficulties.

In Table 2, the comparison of modular multiplication methods for SIDH implementations on 32-bit ARMv7 Cortex-A processors is described. Previous approaches used the fastest multiplication technique in the non-redundant representation. Koziel et al. [2016] and Jalali et al. [2018] utilized COS method from Seo et al. [2016], which is an optimal approach for NEON-only implementation. For the fast computation, Karatsuba technique is applied to COS method. Seo et al. [2018] used HS by integrating ARM and NEON instructions to exploit super-scalar features. The

[4]Reference code is also compiled with `-O3` option.

Table 3. Comparison of Implementations of the SIDHp503 Protocol on ARM Cortex-A5, A7, and A15 (ARMv7-A Architecture) Processors

Implementation	Architecture	Language	Instruction	Timings [cc]		Timings [$cc \times 10^6$]				
				$\mathbb{F}_p$ mul	$\mathbb{F}_p$ sqr	Alice R1	Bob R1	Alice R2	Bob R2	Total
SIDH v3.2 [Costello et al. 2018]	A5	C	ARM	34,743	–	842	926	686	782	3,236
Seo et al. [2018]		ASM	ARM/NEON	2,001	–	100	109	80	92	381
This work		ASM	NEON	1,287	1,190	70	77	56	65	268
SIDH v3.2 [Costello et al. 2018]	A7	C	ARM	30,140	–	795	875	647	732	3,049
Seo et al. [2018]		ASM	ARM/NEON	1,885	–	93	102	75	86	356
This work		ASM	NEON	1,279	1,163	63	70	51	59	243
SIDH v3.2 [Costello et al. 2018]	A15	C	ARM	8,947	–	597	657	487	555	2,296
Koziel et al. [2016]		ASM	NEON	1,372	–	83	87	66	68	302
Seo et al. [2018]		ASM	ARM/NEON	780	–	46	50	38	42	176
This work		ASM	NEON	552	484	29	32	23	27	111

Timings are reported in terms of clock cycles.

Table 4. Comparison of Implementations of the SIKEp503 Protocol on ARM Cortex-A5, A7, and A15 (ARMv7-A Architecture) Processors

Implementation	Architecture	Timings [$cc \times 10^6$]			
		KeyGen	Encaps	Decaps	Total
SIDH v3.2 [Costello et al. 2018]	A5	928	1,529	1,625	3,154
Seo et al. [Seo et al. 2018]		111	183	194	377
This work		78	127	136	263
SIDH v3.2 [Costello et al. 2018]	A7	868	1,432	1,549	2,981
Seo et al. [2018]		103	172	182	354
This work		72	118	126	244
SIDH v3.2 [Costello et al. 2018]	A15	655	1,081	1,149	2,230
Jalali et al. [2018]		68	112	121	233
Seo et al. [2018]		50	83	90	173
This work		32	53	56	109

Timings are reported in terms of clock cycles. Total timing includes encapsulation and decapsulation.

implementation also utilized 1-level Karatsuba algorithm and non-redundant representation. Unlike previous modular multiplication of SIDH protocols, we firstly utilized the redundant representation with NEON instruction sets. The main body of multiplication follows OS method and Karatsuba algorithm. As in Seo et al. [2018], the redundant representation can implement with both ARM and NEON instructions but ARM instruction has limited register space and it is inefficient to handle signed 64-bitwise computations. For this reason, we selected NEON-only instruction.

Table 3 shows the execution timing of finite field arithmetic and SIDHp503 protocol on 32-bit ARM Cortex-A5, A7, and A15 processors. Compared with previous modular multiplication, the proposed modular multiplication on 32-bit ARM Cortex-A15 improves the performance by 93.8%, 59.7%, and 29.2% than SIDH v3.2, Koziel et al., and Seo et al., respectively. Furthermore, we presented the modular squaring method. This enhances the performance by 94.5%, 64.7%, and 37.9% than modular multiplication of SIDH v3.2, Koziel et al., and Seo et al., respectively. The

performance enhancement comes from efficient implementations of multiplication, squaring and Montgomery reduction in the non-redundant representation.

The speed-up of finite field arithmetic directly reflects on the full SIDH protocol. Compared with previous works, SIDHp503 key exchange on 32-bit ARM Cortex-A15 is executed approximately 20.68×, 2.72×, and 1.58× faster than SIDH v3.2, Koziel et al., and Seo et al., respectively. We also evaluated the proposed method on 32-bit ARM Cortex-A5 and A7 processors. Compared with Seo et al.'s work, the modular multiplication is improved by 35.6% and 32.1% for A5 and A7, respectively. The modular squaring operation is also enhanced by 40.5% and 38.3% for A5 and A7, respectively. By optimizing these cryptographic primitives, the performance of SIDHp503 key exchange is optimized by 1.42× and 1.46× for A5 and A7, respectively, than Seo et al.'s work.

In Table 4, implementations of SIKEp503 are given. Proposed implementations achieved the highest performance. Compared with CHES'18 works, the proposed implementation outperforms by 30.2%, 31.0%, and 36.9% for A5, A7, and A15, respectively.

6 CONCLUSION

This article presented NEON-assisted implementations for high-speed finite field arithmetic and SIDHp503/SIKEp503 on 32-bit ARMv7 Cortex-A processors. In particular, we carefully optimized modular multiplication and modular squaring operations in the redundant representation. We integrated our fast modular arithmetic implementations into Microsoft's SIDH library and reported the fastest performance on 32-bit ARM Cortex-A processors to date. A full key-exchange execution of SIKEp503 is performed in about 109 million cycles on 32-bit ARMv7 Cortex-A15 processors (i.e., 54.5 ms @2.0 GHz). The result, which pushes further the performance of post-quantum supersingular isogeny-based protocols, is 1.58× faster than previously fastest assembly-optimized implementations of SIDH on the same processor presented in CHES'18.

REFERENCES

Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. 2019. Supersingular Isogeny Key Encapsulation—Submission to the NIST's Post-Quantum Cryptography Standardization Process, round 2. Retrieved from https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions/SIKE.zip.

Daniel J. Bernstein. 2009. Batch binary edwards. In *Proceedings of the Annual International Cryptology Conference*. Springer, 317–336.

Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. 2014. Curve41417: Karatsuba revisited. In *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES'14)*. Springer, 316–334.

Daniel J. Bernstein and Peter Schwabe. 2012. NEON crypto. In *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES'12)*. Lecture Notes in Computer Science, Vol. 7428, E. Prouff and P. R. Schaumont (Eds.). Springer, 320–339.

Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, and Gregory M. Zaverucha. 2013. Montgomery multiplication using vector instructions. In *Proceedings of the Selected Areas in Cryptography (SAC'13)*. Springer, 471–489.

Craig Costello, Patrick Longa, and Michael Naehrig. 2016. Efficient algorithms for supersingular isogeny diffie-hellman. In *Proceedings of the Advances in Cryptology Conference (CRYPTO'16)*. Lecture Notes in Computer Science, Vol. 9814. Matthew Robshaw and Jonathan Katz (Eds.). Springer, 572–601.

Craig Costello, Patrick Longa, and Michael Naehrig. 2016–2018. SIDH Library. Retrieved from https://github.com/Microsoft/PQCrypto-SIDH.

Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. 2016. On the security of supersingular isogeny cryptosystems. In *Proceedings of Advances in Cryptology: 22nd International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'16)*. 63–91.

Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. 2017. A modular analysis of the fujisaki-okamoto transformation. In *Proceedings of the 15th International Conference on Theory of Cryptography (TCC'17)*. 341–371.

Amir Jalali, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2018. NEON SIKE: Supersingular isogeny key encapsulation on ARMv7. In *Proceedings of the International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 37–51.

Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, and Daivd Jao. 2017. Supersingular isogeny diffie-hellman key exchange on 64-bit ARM. *IEEE Trans. Depend. Sec. Comput.* 16, 5 (2017), 902–912.

David Jao and Luca De Feo. 2011. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Proceedings of the Post-Quantum Cryptography (PQCrypto'11)*, Lecture Notes in Computer Science, Vol. 7071. Bo-Yin Yang (Ed.). Springer, 19–34.

Philipp Koppermann, Eduard Pop, Johann Heyszl, and Georg Sigl. 2018. 18 Seconds to Key Exchange: Limitations of Supersingular Isogeny Diffie-Hellman on Embedded Devices. *Cryptology ePrint Archive, Report 2018/932*. Retrieved from https://eprint.iacr.org/2018/932.

Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2020. SIKE'd Up: Fast hardware architectures for supersingular isogeny key encapsulation. *IEEE Trans. Circ. Syst. I: Regul. Pap.* 67, 12 (2020), 4842–4854.

Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. 2016. NEON-SIDH: Efficient implementation of supersingular isogeny diffie-hellman key exchange protocol on ARM. In *Proceedings of the International Conference on Cryptology and Network Security (CANS'16)*. Springer, 88–103.

Weiqiang Liu, Jian Ni, Zhe Liu, Chunyang Liu, and Máire O'Neill. 2019a. Optimized modular multiplication for supersingular isogeny diffie-hellman. *IEEE Trans. Comput.* 68, 8 (2019), 1249–1255.

Weiqiang Liu, Ziying Ni, Jian Ni, Ciara Rafferty, and Máire O'Neill. 2019b. High performance modular multiplication for SIDH. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 39, 10 (2019), 3118–3122.

Patrick Longa. 2016. Four$\mathbb{Q}$ NEON: Faster elliptic curve scalar multiplications on ARM processors. In *Proceedings of the International Conference on Selected Areas in Cryptography*. Springer, 501–519.

Paulo Martins and Leonel Sousa. 2014. On the evaluation of multi-core systems with SIMD engines for public-key cryptography. In *Proceedings of the Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW'14)*. IEEE, 48–53.

Paulo Martins and Leonel Sousa. 2015. Stretching the limits of programmable embedded devices for public-key cryptography. In *Proceedings of the Workshop on Cryptography and Security in Computing Systems*. ACM, 19.

Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44, 170 (1985), 519–521.

NIST. 2017–2019. Post-Quantum Cryptography Standardization. Retrieved from https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

Krishna Chaitanya Pabbuleti, Deepak Hanamant Mane, Avinash Desai, Curt Albert, and Patrick Schaumont. 2013. SIMD acceleration of modular arithmetic on contemporary embedded platforms. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'13)*. IEEE, 1–6.

Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. 2020. Supersingular isogeny key encapsulation (SIKE) round 2 on ARM Cortex-M4.*IACR Cryptol. IEEE Transactions on Computers*. Early Access.

Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. 2019b. Optimized SIKE round 2 on 64-bit ARM. In *Proceedings of the World Conference on Information Security Applications (WISA'19)*. Springer.

Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. 2019a. SIKE round 2 speed record on ARM Cortex-M4. In *Proceedings of the International Conference on Cryptology and Network Security*. Springer, 39–60.

Hwajeong Seo, Zhe Liu, Johann Großschädl, Jongseok Choi, and Howon Kim. 2014. Montgomery modular multiplication on ARM-NEON revisited. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 328–342.

Hwajeong Seo, Zhe Liu, Johann Großschädl, and Howon Kim. 2016. Efficient arithmetic on ARM-NEON and its application for high-speed RSA implementation. *Secur. Commun. Netw.* 9, 18 (2016), 5401–5411.

Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. 2018. SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 1, 3 (2018), 1–20.

Hwajeong Seo, Zhe Liu, Yasuyuki Nogami, Taehwan Park, Jongseok Choi, Lu Zhou, and Howon Kim. 2015. Faster ECC over $\mathbb{F}_{2^{521}-1}$(feat. NEON). In *Proceedings of the Annual International Conference on Information Security and Cryptology (ICISC'15)*. Springer, 169–181.

Peter W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE, 124–134.

Received June 2020; revised November 2020; accepted November 2020