Evaluation of modular multiplication techniques for Supersingular Isogeny Schemes on ARMv8 cores

Vitor Satoru Machi Matsumine[1], Félix Carvalho Rodrigues[1], Décio Gazzoni Filho[1], Caio Teixeira[1], Julio López[1], Ricardo Dahab[1]

[1] Institute of Computing – University of Campinas (Unicamp)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

{vitor.matsumine,caio.teixeira}@students.ic.unicamp.br
{felix.rodrigues,ra264965,jlopez,rdahab}@ic.unicamp.br

Abstract. *This paper focuses on the evaluation of different modular multiplication implementation techniques on 64-bit ARMv8 systems for the third-round NIST alternate candidate SIKE. The benchmarks were performed on four devices: an Orange Pi WinPlus featuring the Cortex-A53 processor, an NVIDIA Jetson Nano with a Cortex-A57, a Raspberry Pi 4 with a Cortex-A72 and a Macbook Air based on an Apple M1 chip. Throughout these platforms we observed that the two-level Karatsuba Comba method performs better on most Cortex-A processors but the Operand Scanning method presented a performance improvement ranging from* 10% *to* 43% *for the multiplication procedure and a* 7% *to* 25% *improvement for the modular reduction on the Apple M1 for all SIKE security levels, resulting in an overall improvement ranging from* 8% *to* 28% *for the SIKE KEM operations on this architecture.*

1. Introduction

Recent developments in quantum computing brought to attention the possibility that the current standards in public-key cryptography, based on the hardness of integer factorization and discrete logarithms may be broken in a near future by a sufficiently large quantum computer running Shor's algorithm [Shor 1999]. This led the cryptography community to seek new mathematical problems that could be used for public-key cryptography resistant to both classical and quantum cryptanalysis, and that could be implemented in our current classical computers, giving rise to the field of post-quantum cryptography.

In this context, we have the National Institute of Standards and Technology (NIST) Post-Quantum Standardization Process [NIST 2017], which seeks to set new standards for post-quantum cryptography. Currently in its third round, this competition has the Supersingular Isogeny Key Encapsulation (SIKE) protocol as one of its alternate candidates. SIKE has its origins on the Supersingular Isogeny Diffie-Hellman (SIDH) scheme [De Feo et al. 2014] and is based on the hardness of computing isogenies between supersingular elliptic curves.

Since its initial inception, several improvements have been made to the SIKE protocol and in this paper we highlight the improvements made to the ARMv8 implementation of SIKE [Jalali et al. 2019, Seo et al. 2020]. These improvements have been focused mainly in the base field arithmetic operations, with its most costly operation being modular multiplication, consisting in multi-precision multiplication followed by modular reduction.

Throughout the history of improvements in modular multiplication [Jalali et al. 2019, Seo et al. 2020], special attention was brought to the multiplication techniques used for each security level, with the use of Operand Scanning and Product Scanning techniques, as well as the use of the Karatsuba multiplication. These different techniques are applied to the multi-precision multiplication procedure and the internal multiplications of the modular reduction

In this work we investigate the software implementation of these multiplication techniques when applied to the SIKE protocol on different ARMv8 platforms, including members of the Cortex-A family and the more recent Apple M1. For each ARMv8 platform, we compare our optimized hand-written assembly implementation with the state-of-the-art reference implementation of SIKE.

In light of a recently discovered efficient key recovery attack on SIDH [Castryck and Decru 2022], the current NIST submission of SIKE may be considered insecure. However, the results and discussions in this paper are still applicable to other schemes that utilize modular multiplication over large prime characteristic fields.

1.1. Our contributions

In this paper we evaluated the multi-precision multiplication techniques based on Operand Scanning, Product Scanning and Karatsuba multiplication for the different security parameters of the NIST PQC competition's third-round alternate candidate SIKE on different 64-bit ARMv8 platforms. We developed hand-written A64 assembly implementations for the multi-precision multiplication and reduction procedures of the base field arithmetic for the SIKE protocol. The first implementation consists in the multiplication with Operand Scanning. The second is the multiplication with two-level Karatsuba Comba (Product Scanning) with a different instruction interleaving when compared to the current state-of-the-art of the SIKE implementation for ARMv8 (present in the Microsoft PQCrypto-SIDH 3.5 version) and the use of SIMD registers as cache memory to substitute intermediate memory operations. For the third implementation, we optimize Montgomery reduction for the SIKE protocol with Operand Scanning for its internal multiplications.

Our source code was compiled with ARMv8 Clang and GCC compilers and its performance was evaluated on an Orange Pi WinPlus (Cortex-A53), an NVIDIA Jetson Nano (Cortex-A57), a Raspberry Pi 4 (Cortex-A72) and a Macbook Air based on an Apple M1 chip. Our benchmarks show that:

1. on the Cortex-A family: the two-level Karatsuba Comba method was the fastest throughout most processors;
2. on the Apple M1: the Operand Scanning method was the fastest;
3. on Cortex-A53: Montgomery reduction with Operand Scanning was the fastest;
4. on the Apple M1: Montgomery reduction with Operand Scanning was the fastest.

For the SIKE KEM operations on the Apple M1 with Operand Scanning for multiplication and Montgomery reduction, we also observed an improvement of 28% for SIKEp434, 8% for SIKEp503, 14% for SIKEp610 and 8% for SIKEp751.

1.2. Paper organization

Section 2 presents a brief description of the SIDH and SIKE protocols. Section 3 introduces the ARMv8 architecture and the cores targeted by our experiments. Section

4 describes the implementation techniques for multi-precision multiplication and Montgomery reduction. Section 5 describes our experiments and presents benchmarks for our implementations. Section 6 presents our concluding remarks.

2. Isogeny Based Cryptography

Isogeny-based cryptography has its roots in Elliptic Curve Cryptography (ECC), with its first reported instance being presented in 1997 by Couveignes in his notes on "Hard Homogeneous Spaces" which were rejected by CRYPTO97 and later published in 2006 [Couveignes 2006]. It would be later independently rediscovered by Rostovtsev and Stolbunov in [Rostovtsev and Stolbunov 2006]. These schemes were based on isogenies between ordinary curves and presented significant performance issues.

The development of subexponential-time quantum algorithms capable of computing isogenies between ordinary elliptic curves motivated the work of [De Feo et al. 2014] on the development of a Diffie-Hellman-like scheme relying on the conjectured difficulty of finding isogenies between supersingular elliptic curves. The resulting scheme, named Supersingular Isogeny Diffie-Hellman (SIDH), brought significant performance improvements to isogeny schemes, as well as security improvements, with the fastest quantum attack remaining exponential. Since its initial proposal, several performance improvements, in regards to both its running time and key size, were made, which resulted in the Supersingular Isogeny Key Encapsulation (SIKE), a Key Encapsulation Mechanism (KEM) based on SIDH.

2.1. SIDH

The SIDH protocol makes use of primes of the form $p = l_A^{e_A} l_B^{e_B} \cdot f \pm 1$ for small primes l_A and l_B, positive integers e_A and e_B, and a small cofactor f. For this prime p we define the extension field $\mathbb{F}_{p^2}$ from which the initial supersingular elliptic curve E with cardinality $\#E = (l_A^{e_A} l_B^{e_B} \cdot f)^2$ is defined. On this curve, we denote by P_A, Q_A and P_B, Q_B the basis points that generate, respectively, the torsion subgroups $\langle P_A, Q_A \rangle = E[l_A^{e_A}]$ and $\langle P_B, Q_B \rangle = E[l_B^{e_B}]$.

On the SIDH key exchange, Alice takes a random integer $m_A \in \mathbb{Z}/l_A^{e_A}\mathbb{Z}$ as her secret key and calculates the point $R_A = P_A + [m_A]Q_A$, which is then used to calculate the isogeny $\phi_A : E \to E_A$ with $\langle R_A \rangle$ as its kernel. Alice then evaluates the points P_B and Q_B through her isogeny ϕ_A and publishes her public key $\{\phi_A(P_B), \phi_A(Q_B), E_A\}$. Similarly, Bob takes a random integer $m_B \in \mathbb{Z}/l_B^{e_B}\mathbb{Z}$, calculates the point $R_B = P_B + [m_B]Q_B$ and the isogeny $\phi_B : E \to E_B$, evaluates the points P_A and Q_A through ϕ_B and publishes his public key $\{\phi_B(P_A), \phi_B(Q_A), E_B\}$.

Once Alice obtains Bob's public key, she calculates the point $R_{AB} = \phi_B(P_A) + [m_A]\phi_B(Q_A)$ which is used to calculate the isogeny $\phi_{AB} : E_B \to E_{AB}$ with kernel $\langle R_{AB} \rangle$. Alice then takes the j-invariant of the curve E_{AB} as her shared key. Similarly, Bob takes Alice's public key to calculate the point $R_{BA} = \phi_A(P_B) + [m_B]\phi_A(Q_B)$, calculates the isogeny $\phi_{BA} : E_A \to E_{BA}$ with kernel $\langle R_{BA} \rangle$, and takes the j-invariant of curve E_{BA} as his shared key with Alice. Since the curves E_{AB} and E_{BA} are isomorphic, they produce the same j-invariant and thus this value can be used as the secret shared key between Alice and Bob.

2.2. SIKE

The Supersingular Isogeny Key Encapsulation protocol is based on the Public Key Exchange (PKE) obtained from SIDH [De Feo et al. 2014] with the application of the Hofheinz, Hövelmanns and Kiltz transform [Hofheinz et al. 2017] in order to obtain an actively secure IND-CCA KEM.

In a similar manner to SIDH, SIKE takes a prime of the form $p = 2^{e_A} \cdot 3^{e_B} - 1$ which is used to define the extension field $\mathbb{F}_{p^2}$. The initial supersingular curve is defined as $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + 6x^2 + x$, where E_0 has cardinality $(2^{e_A} \cdot 3^{e_B})^2$. The public basis points are defined as $\langle P_A, Q_A \rangle = E_0[2^{e_A}]$ and $\langle P_B, Q_B \rangle = E_0[3^{e_B}]$. SIKE's key encapsulation mechanism is based on three procedures: key generation, encapsulation and decapsulation.

Key generation: Alice takes a secret random integer $sk_A \in \mathbb{Z}/2^{e_A}\mathbb{Z}$ and a random bitstring $s \in \{0,1\}^t$. She then calculates the isogeny $\phi_A : E_0 \rightarrow E_A$ with kernel $\langle P_A + [sk_A]Q_A \rangle$ and computes her public key $pk_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$.

Encapsulation: Bob generates a random message $m \in \{0,1\}^t$, concatenates m with pk_A and hashes $m||pk_A$ with SHAKE256 obtaining a hash digest r. Bob then uses r as his secret key sk_B, calculates the isogeny $\phi_B : E_0 \rightarrow E_B$ with kernel $\langle P_B + [sk_B]Q_B \rangle$ and generates his public key $c_0 = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$. Bob then uses Alice's public key pk_A to calculate the isogeny $\phi_{BA} : E_A \rightarrow E_{BA}$ with kernel $\langle \phi_A(P_B) + [sk_B]\phi_A(Q_B) \rangle$ and takes the j-invariant of curve E_{BA}. This j-invariant is then hashed with SHAKE256 and the digest h is obtained. The message m is then XORed with h to obtain c_1. Lastly, Bob concatenates m with (c_0, c_1) and hashes the result with SHAKE256 to obtain the digest K, which will be used as the session key. The tuple (c_0, c_1) is then sent to Alice.

Decapsulation: Alice takes Bob's public key c_0 and uses it to calculate the isogeny $\phi_{AB} : E_B \rightarrow E_{AB}$ with kernel $\langle \phi_B(P_A) + [sk_A]\phi_B(Q_A) \rangle$, takes the j-invariant of E_{AB} and hashes it with SHAKE256 to obtain the digest h. Alice then obtains the message m' by XOR'ing h with c_1, and validates Bob's public key by concatenating m with her public key pk_A, arriving at the secret value r'. Alice then proceeds to calculate the isogeny $\phi'_B : E_0 \rightarrow E'_B$ with kernel $\langle P_B + [r']Q_B \rangle$ and generate the public key $c'_0 = \{E'_B, \phi_B(P_A)', \phi_B(Q_A)'\}$. If the calculated c'_0 matches the received c_0, then the public key is considered valid and the session key K is calculated by hashing m' concatenated with (c_0, c_1) using SHAKE256. Otherwise, Alice hashes s (obtained in the key generation phase) concatenated with (c_0, c_1), thus invalidating the session key K.

Security parameters: SIKE's security parameters are based on the size of the prime p that defines the finite field $\mathbb{F}_{p^2}$ and consequently defines the cardinality $\#E = (2^{e_A} \cdot 3^{e_B})^2$ of the supersingular elliptic curves employed in the protocol. Consequently, they also define the size of the torsion subgroups $E[2^{e_A}]$ and $E[3^{e_B}]$ from which the secret points that generate the kernel for the secret isogeny are derived. On Table 1 we show the different SIKE security levels, with the size of the prime p in bits being given by "`SIKEp<bitlenght_of_p>`", in addition to their corresponding NIST security level and the number of 64-bit words needed to represent elements of the base prime field $\mathbb{F}_p$ for each security level.

Efficiency aspects: since the initial proposal of the SIDH protocol and the current version of SIKE, several improvements have been made in order to optimize the per-

Table 1. SIKE parameters

Prime	NIST Security Level	Number of 64-bit words
SIKEp434	Level I	7
SIKEp503	Level II	8
SIKEp610	Level III	10
SIKEp751	Level V	12

formance of the protocol. Here we highlight the following contributions: use of x-coordinate-only arithmetic with Montgomery curves for both point arithmetic and isogeny operations [Costello et al. 2016]; optimized Montgomery reduction and the three-point ladder from [Faz-Hernández et al. 2017]; and the optimized SIKE implementations for ARMv8 [Koziel et al. 2016, Jalali et al. 2017, Jalali et al. 2019, Seo et al. 2020], which were focused on writing optimized hand-written assembly implementations of the base field arithmetic on $\mathbb{F}_p$, with a special regard to the multi-precision multiplication techniques and the modular reduction implementation.

3. ARMv8 Architecture

In 2011, the ARMv8 architecture added support for 64-bit operations to the ARM family of processors. This architecture is known as AArch64, while its instruction set is called A64. We will analyze the performance of the Cortex-A family of processors and the Apple M1 based on this architecture for our implementation of the multi-precision multiplication techniques. In this section we introduce the target cores for our experiments and highlight some of the instructions used for building our implementations.

3.1. Target cores

The Cortex-A53 is one of the first ARM cores to implement the ARMv8 architecture. It is a power-efficient 2-way superscalar processor with an in-order, 8-stage, dual-issue pipeline featuring 8 to 64 KB of L1 cache memory and 128 KB to 2 MB of L2 cache memory.

Another target core for our implementations is the Cortex-A57. A more powerful alternative to the Cortex-A53, the Cortex-A57 features the ARMv8 architecture in a 3-way superscalar processor with an out-of-order, speculative execution pipeline with 48 KB of L1 I-cache memory, 32 KB of L1 D-cache memory and 512 KB to 2 MB of L2 cache memory.

In the higher-end of the Cortex-A family of processors we have the Cortex-A72, which was designed as a successor to the Cortex-A57 core with improved power efficiency and performance. This processor features an out-of-order, speculative execution 3-way superscalar execution pipeline and includes 48 KB of L1 I-cache memory, 32 to 64 KB of L1 D-cache memory and 512 KB to 4 MB of L2-cache.

Our final target of interest is the M1, a new ARMv8-based system on a chip (SoC) developed by Apple. It features an unusually large 8-way superscalar pipeline with out-of-order execution and a reorder buffer size conjectured to exceed 600 entries, while maintaining significant power efficiency. It features eight cores, evenly split between performance and efficiency cores. It implements the ARMv8.4-A instruction set and features L1 cache memories of 192 KB (instruction) and 128 KB (data) per performance core and

128 KB (instruction) and 64 KB (data) per efficiency core, as well as shared L2 cache memories of 12 MB for the performance cores and 4 MB for the efficiency cores.

3.2. A64 Instructions

Here we highlight and describe some of the main A64 instructions used in our implementations of the multi-precision multiplication procedures:

1. `MUL X0, X1, X2` [$X_0 \leftarrow L(X_1 \times X_2)$]. Unsigned multiplication, low part;
2. `UMULH X0, X1, X2` [$X_0 \leftarrow H(X_1 \times X_2)$]. Unsigned multiplication, high part;
3. `ADD X0, X1, X2` [$X_0 \leftarrow X_1 + X_2$]. Unsigned addition. To raise potential carry flags during the addition, we use the `ADDS` instruction; to perform the addition with a carry we use the `ADC` instruction; to perform both simultaneously, we use the `ADCS` instruction;
4. `SUB X0, X1, X2` [$X_0 \leftarrow X_1 - X_2$]. Unsigned subtraction. Following a similar pattern for the borrow handling as with the addition carry flag, we have the instructions `SUBS`, `SBC` and `SBCS`.
5. `MOV X0, Vd.d[index]`. Moves the 64-bit value present in the scalar register X_0 to the SIMD register `Vd` on a specific lane `index`. The instruction `MOV Vd.d[index], X0` can be used to move the value present in `Vd.d[index]` back to the X_0 register;

4. Implementation Techniques

In this section, we will describe the multi-precision multiplication implementation techniques that are used in our implementations of multiplication and modular reduction procedures for the different SIKE parameters on ARMv8. For the descriptions, we consider $C = A \cdot B = (C[2n-1], \cdots, C[1], C[0])$ as the product of two n-words operands $A = (A[n-1], \cdots, A[1], A[0])$ and $B = (B[n-1], \cdots, B[1], B[0])$.

4.1. Karatsuba multiplication

The Karatsuba method is a multiplication technique in which a multiplication of two m-bit operands is replaced by three multiplications of $(m/2)$-bit operands and some addition and subtraction operations. Using this method, a product $C = A \cdot B$, where A and B are n-words operands can be performed by first splitting A into low and high parts, given by $A = A_H \cdot 2^{n/2} + A_L$, and similarly for B. The multiplication can then be performed through the following equation for the additive (1) and subtractive (2) formulations of Karatsuba multiplication:

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

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

This technique can then be further applied on the intermediate multiplications, giving rise to the two-level Karatsuba multiplication.

Implementation: to exemplify the implementation, we will focus on the additive Karatsuba multiplication. We begin by calculating the sums $(A_H + A_L)$ and $(B_H + B_L)$, and saving the carry bits c_A and c_B that may be generated by each. These carry bits could

generate an additional word that would involve a costly additional iteration in the multiplication of $(A_H + A_L) \cdot (B_H + B_L)$; however, since this additional multiplication would be either a product of $2^{n/2} \cdot (A_H + A_L)$, $2^{n/2} \cdot (B_H + B_L)$ or the sum of both, we can instead conditionally add $(A_H + A_L)$, $(B_H + B_L)$ or $(A_H + A_L) + (B_H + B_L)$ shifted by $2^{n/2}$ words given the carry bits c_A and c_B as follows:

1. $c_A, A_{H+L} \leftarrow A_H + A_L$
2. $c_B, B_{H+L} \leftarrow B_H + B_L$
3. $AB_{H+L} \leftarrow (A_H + A_L) \cdot (B_H + B_L)$
4. if $c_A = 1$ and $c_B = 1$, then $AB_{H+L} \leftarrow AB_{H+L} + 2^{\frac{n}{2}} \cdot (A_{H+L} + B_{H+L})$
5. else if $c_A = 1$, then $AB_{H+L} \leftarrow AB_{H+L} + 2^{\frac{n}{2}} \cdot (B_{H+L})$
6. else if $c_B = 1$, then $AB_{H+L} \leftarrow AB_{H+L} + 2^{\frac{n}{2}} \cdot (A_{H+L})$

To prevent side-channel timing attacks, this procedure must be performed in constant time. For that end, in order to avoid conditional statements that could result in a variable execution time, we calculate conditional masks from the values of c_A and c_B and apply them to the words of $A_{H+L} + B_{H+L}$:

```
1 //mask calculation
2 sub M_B, xzr, c_A
3 sub M_A, xzr, c_B
```

```
1 //word masking
2 and A_HL_0, A_HL_0, M_A
3 and A_HL_1, A_HL_1, M_A
4 ...
5 and A_HL_n/2, A_HL_n/2, M_A
6 and B_HL_0, B_HL_0, M_B
7 and B_HL_1, B_HL_1, M_B
8 ...
9 and B_HL_n/2, B_HL_n/2, M_B
```

We then combine the carry bits and add the masked values together. The carry bit of this sum is then added to the combined carry bits:

```
1 //combined carry
2 and c_AB, c_A, c_B
```

```
1 //add masked values
2 adds sum_AB_0, A_HL_0, B_HL_0
3 adcs sum_AB_1, A_HL_0, B_HL_0
4 ...
5 adcs sum_AB_n/2, A_HL_n/2, B_HL_n/2
6 adc  c_AB, c_AB, xzr
```

This procedure has the following effect: if a carry bit has value 1, the resulting mask has value `0xfff...ff`, therefore the masked value would remain unchanged. Otherwise, if a carry bit has value 0, the resulting mask has value `0x000...00` and the masked value would be set to 0. By adding these masked values of A_{H+L} and B_{H+L} together, the resulting sum would be as follows

1. $c_A = 1, c_B = 1 \rightarrow \texttt{sum\_}AB_{H+L} = A_{H+L} + B_{H+L}$
2. $c_A = 1, c_B = 0 \rightarrow \texttt{sum\_}AB_{H+L} = B_{H+L}$
3. $c_A = 0, c_B = 1 \rightarrow \texttt{sum\_}AB_{H+L} = A_{H+L}$
4. $c_A = 0, c_B = 0 \rightarrow \texttt{sum\_}AB_{H+L} = 0$

We can then proceed to calculate $(A_H + A_L) \cdot (B_H + B_L)$ and add the masked sum shifted by $2^{n/2}$ words to this result. We then conclude this calculation by adding the carry of this sum to the combined carry c_{AB}.

Then, we calculate the product $A_L \cdot B_L$. The lowest order $n/2$ words of the result consist in the $n/2$ lowest order words of the full product C and can be directly stored. We then compute the subtraction $(A_H + A_L) \cdot (B_H + B_L) - (A_L \cdot B_L)$ and subtract a possible borrow from this calculation from c_{AB}. Next, we calculate the product $A_H \cdot B_H$, compute $(A_H + A_L) \cdot (B_H + B_L) - (A_L \cdot B_L) - (A_H \cdot B_H)$ and subtract a possible borrow from this calculation from c_{AB}.

Finishing our calculations, we add the $n/2$ highest order words of $A_L \cdot B_L$ to the $n/2$ lowest order words of $(A_H + A_L) \cdot (B_H + B_L)$, and add the $n/2$ lowest order words of $A_H \cdot B_H$ to the $n/2$ highest order words of $(A_H + A_L) \cdot (B_H + B_L)$. We also must adequately handle the carry propagation between these sums and also add the combined carry c_{AB} to the next word after $(A_H + A_L) \cdot (B_H + B_L)$. With this, the full product C is calculated using additive Karatsuba multiplication.

4.2. Operand Scanning

The Operand Scanning method (also referred as Schoolbook or Row-wise) is a straightforward way of performing large integer multiplication. In this method we fix a word $B[j]$ and iterate through the words of A while performing the multiplication $A[i] \times B[j]$, for $0 \leq i, j < n$, saving the intermediate values and adding them to the appropriate accumulators corresponding to the words of C. The result of the $A[i] \times B[j]$ multiplication consists in a low part $L(A[i] \times B[j])$ and a high part $H(A[i] \times B[j])$ (respectively, the first and second words of the full result). The low part must be added to the accumulator for the word $C[i + j]$ and the high part must be added to the accumulator for the word $C[i + j + 1]$.

In this method, we load the n words of A and keep all of them in registers, load at least one word of B for the current iteration and store the intermediate results in n registers that are stored in a ring buffer; i.e., once the current lowest order word of C is calculated, the result is stored and the corresponding accumulator is used to store the result of the next highest order word of C.

Implementation: in our Operand Scanning implementation, we sought to maximize the instruction level parallelism by interleaving multiplication instructions with addition and memory accesses while minimizing the register dependency between these instructions. For SIKEp434's multi-precision multiplication, elements of $\mathbb{F}_p$ are unsigned integers, each being represented by seven 64-bit words. Thus, the elements A and B of the multiplication are represented as $A = (A_6, A_5, \cdots, A_0)$ and $B = (B_6, B_5, \cdots, B_0)$, with A_6 representing the word with the most significant bits and A_0 the least significant bits. We begin the multiplication as follows.

For the `p434` multiplication with Operand Scanning, we use one register for the current B operand, seven registers for the words $[A_0, \cdots, A_6]$, three registers for the partial results of the multiplication $[\texttt{PART0}, \cdots, \texttt{PART2}]$ and seven registers that are used as the accumulators for the values of C named $[\texttt{ACC0}, \cdots, \texttt{ACC6}]$. These accumulators are then cycled once a calculation of a word of C is finished.

For this procedure, we begin by loading B_0 into the register B. This register will

```
ldr B, [ADDR_B0] // B <- B0
sub sp, sp, (8*N_REG)
ldp A0, A1, [ADDR_A0]
str PART0, [sp, 0]
mul ACC0, A0, B // ACC0 <- L(A0 x B0)
ldp A2, A3, [ADDR_A2]
mul ACC1, A1, B // ACC1 <- L(A1 x B0)
```

be used to store the word of B being processed during the current iteration of the multiplication. We then proceed to subtract $8 \cdot$ N_REG, where N_REG is the number of registers we want free, ranging from x19 to x30, and store the contents of this register into the stack, as these registers need to be returned to their original state when the function is finished. We interleave this procedure and the loading of the next words of A with the beginning of the multiplication $A \cdot B$, given by the low parts of $A_0 \times B_0$ and $A_1 \times B_0$, named $L(A_0 \times B_0)$ and $L(A_1 \times B_0)$.

```
mul PART0, B1, A0       // PART0 <- L(A0 X B1)
mul PART1, B1, A1       // PART1 <- L(A1 X B1)
adds ACC1, ACC1, PART0  // C[1] <- C[1] + L(A0 X B1)
mul PART2, B1, A2       // PART2 <- L(A1 X B2)
adcs ACC2, ACC2, PART1  // C[2] <- C[2] + L(A1 X B1) + carry
```

We seek to maintain the interleaving of multiplication instructions with addition instructions whenever possible, while also minimizing the register dependencies that could cause a pipeline stall. Observe that in this step of the computation, $C_1 = H(A_0 \times B_0) + L(A_1 \times B_0) + L(A_0 \times B_1)$ finishes calculating once $L(A_0 \times B_1)$ is accumulated into ACC1.

```
mul PART0, B1, A6      // PART0 <- L(A6 X B1)
adcs ACC6, ACC6, PART2 // C[6] <- C[6] + L(A5 X B1)  + carry
str ACC1, [ADDR_C, 8]   // store C[1]
umulh PART1, B1, A0  // x18 <- H(A0 X B1)
```

Since C_1 was fully calculated, we interleave the storing of this word with the remaining calculations.

```
umulh PART0, B1, A5     // x2 <- H(A5 X B1)
adcs ACC6, ACC6, PART2  // C[6] <- C[6] + H(A4 X B1) + carry
ldr B2, [ADDR_B, 16]    // load B2
adcs ACC0, ACC0, PART0  // C[7] <- C[7] + H(A5 X B1) + carry
```

When all the calculations involving a word of B are finished, we replace it while interleaving the load operation with arithmetic operations. We repeat this pattern of interleaving multiplies, additions and memory accesses throughout the remaining iterations of the multi-precision multiplication, as well as the cycling between the accumulator registers, until the full product C is calculated.

4.3. Product Scanning

The Product Scanning method, also known as Comba or Column-wise multiplication, is a multi-precision multiplication method that aims to reduce the number of intermediate results that need to be stored throughout the calculation.

In this method, we prioritize the calculation of all the intermediate products that are part of a word $C[i]$ (for $0 \leq i < n$) and immediately add the results on an accumulator corresponding to the word $C[i]$. The carry bits generated during the accumulation are added to the accumulator that corresponds to the word $C[i+1]$. Once $C[i]$ is fully calculated, the result is stored and we proceed in a similar manner to calculate and accumulate the intermediate products of the word $C[i+1]$.

Implementation: for the implementation of the Product Scanning method, we exemplify it with a small instance using two word elements $A = [A_1, A_0]$ and $B = [B_1, B_0]$ for a product $C = A \cdot B = [C_3, C_2, C_1, C_0]$. The product scanning method focuses on fully calculating the words of C in order to reduce the number of intermediate accumulators for these words. We can see that each word of C consists of the following sums:

$$
\begin{aligned}
C_0 &= L(A_0 \times B_0), \\
C_1 &= H(A_0 \times B_0) + L(A_1 \times B_0) + L(A_0 \times B_1), \\
C_2 &= H(A_1 \times B_0) + H(A_0 \times B_1) + L(A_1 \times B_1), \\
C_3 &= H(A_1 \times B_1).
\end{aligned}
$$

We start by loading the operands and performing the initial multiplications.

```
// loading operands
ldp A0, A1, [ADDR_A, 0]
ldp B0, B1, [ADDR_B, 0]
```

```
//first multiplications
mul C0, A0, B0
mul C1, A1, B0
umulh C2, A1, B0
umulh C3, A1, B1
```

We then proceed to store the first word C_0 that was already fully calculated and continue with the calculation of C_1, while we calculate the partial multiplications for C_2. The word C_1 can then be stored as we progress to the calculation of C_2 and propagate the carry bits to C_3, finishing the calculation.

```
// store C0 and calculate C1
mul TMP0, A0, B1
umulh TMP1, A0, B0
adds C1, C1, TMP0
str C0, [ADDR_C]
adc C2, C2, xzr
mul TMP0, A1, B1
adds C1, C1, TMP1
adc C2, C2, xzr
umulh TMP1, A0, B1
```

```
// calculate C2 and propagate
// carry
adds C2, C2, TMP0
adc C3, C3, xzr
str C1, [ADDR_C, 8]
adds C2, C2, TMP1
adc C3, C3, xzr
stp C2, C3, [ADDR_C, 16]
```

4.4. Modular reduction

To perform a modular multiplication over a field $\mathbb{F}_p$, the multiplied values must be reduced modulo p through a modular reduction. This procedure is often done in two manners: interleaved reduction or lazy-reduction. For the interleaved reduction, we perform the modular reduction on the intermediate products during the calculation of the multiplication and we obtain the fully reduced value at the end. For the lazy-reduction, the full multiplication product is calculated first and then reduced. This method often produces savings in the modular multiplication, as the number of registers are limited and the interleaved method often demands more memory accesses to retrieve the modulus of the reduction.

Implementation: in our implementation, we used Montgomery reduction as our modular reduction method with lazy-reduction. The implementation follows the current most efficient method for computing the modular reduction in SIKE, given by the REDC algorithm for a λ-Montgomery-friendly modulus presented in [Faz-Hernández et al. 2017] with the shifted modulus representation of [Seo et al. 2020]. For the internal multiplications, we used the Operand Scanning method following the instruction interleaving presented in Section 4.2.

Taking SIKEp434's modular reduction as an example, its prime p is a 3-Montgomery-friendly prime. We chose $B = 3$ for the REDC algorithm and we perform two 192x256-bit multiplications and one 64x256-bit multiplication, shifting the position of the 64-bit operand by two words. These multiplications were implemented using the Operand Scanning method, while also interleaving multiplies with additions to the intermediate result T. For p434's reduction, we obtained better results without the use of the shifted representation. For higher security levels, we used the shifted representation and also interleaved the shift correction of intermediate products with the load operations of the operands for the next intermediate multiplication.

5. Results

For each SIKE security parameter set, we implemented the different techniques for multi-precision multiplication and collected benchmarks on four different platforms, representing a range of ARMv8 cores. In this section we describe our experiments, present benchmark results for our implementations of multi-precision multiplication and Montgomery reduction, and analyze our results.

5.1. Experiments

For each set of SIKE security parameters, we implemented two handcrafted A64 assembly procedures for the multi-precision multiplication, one implementing the Operand Scanning method and the other implementing the two-level Karatsuba Comba (Product Scanning) method, both using the instruction interleaving technique described in Section 4. We compare our implementation with the current SIKE reference implementation for ARMv8, present in the Microsoft PQCrypto-SIDH 3.5 version library, which also implements the two-level Karatsuba Comba method. Our implementation of this method differs in the instruction interleaving methods and the use of the `mov` instruction to SIMD registers as a substitute to some intermediate `store` and `load` routines.

For the Montgomery reduction implementation, we used the Operand Scanning method for the internal multiplications as presented in Section 4 and compare our implementation with the reference reduction implementation present in PQCrypto-SIDH, which uses the Comba method for its internal multiplications.

The performance of our implementations were measured using the Google Benchmark framework on four different devices: an Orange Pi WinPlus with four ARM Cortex-A53 running at a clock frequency of 1008 MHz; an NVIDIA Jetson Nano with four ARM Cortex-A57 running at 1479 MHz; a Raspberry Pi 4 with four ARM Cortex-A72 running at 1500 MHz; and an Apple Macbook Air featuring the Apple M1 chip, with eight cores (four performance cores and four efficiency cores) with a maximum CPU clock rate of 3.2 GHz.

Table 2. Multi-precision multiplication benchmarks

	Core	Operand Scanning (ns)	2-level Karatsuba Comba (ns)	Reference (ns)		Operand Scanning (ns)	2-level Karatsuba Comba (ns)	Reference (ns)
p434	**A53**	385	359	364	**p503**	512	413	416
	A57	232	182	182		304	210	211
	A72	229	176	180		299	206	208
	M1	15.8	21.7	22.7		20.7	25.6	25.7
p610	**A53**	778	744	742	**p751**	1195	987	1014
	A57	474	375	376		697	477	478
	A72	467	366	366		673	461	464
	M1	32.2	39.0	38.7		46.6	51.8	51.7

Table 3. Modular reduction benchmarks

	Core	Operand Scanning (ns)	Reference (ns)		Operand Scanning (ns)	Reference (ns)
p434	**A53**	249	271	**p503**	319	342
	A57	133	133		167	167
	A72	131	131		165	162
	M1	9.46	11.9		16.3	17.7
p610	**A53**	457	514	**p751**	702	751
	A57	261	257		368	376
	A72	257	252		361	366
	M1	24.0	25.8		32	35.2

For our benchmarks, we compiled our implementations of the multiplication and reduction procedures alongside the ARMv8 field arithmetic implementation from PQCrypto-SIDH on each target platform. We benchmarked the calls to the multiplication procedures with both implemented techniques and the reference implementation, and present the results on Table 2. For the reduction procedure, we benchmarked our reduction implementation with Operand Scanning and the reference implementation, presenting our results on on Table 3. In the Apple M1, we also benchmarked our implementation of the SIKE KEM with Operand Scanning for both the multiplication and reduction procedures and compared it to the PQCrypto-SIDH implementation, presenting our results on Table 4.

5.2. Discussion

For the multi-precision multiplication, we can observe that across all security levels in the Apple M1, the Operand Scanning technique results in the best performance with speedups of 43%, 24%, 20% and 10% for SIKEp434, SIKEp503, SIKE p610 and SIKEp751, respectively, when compared to the reference implementation. We can also observe a similar trend for the Montogmery reduction with internal multiplications using the Operand Scanning technique in the Apple M1, with a 25% performance improvement for SIKEp434, 8% for SIKEp503, 7% for SIKEp610 and 10% for SIKEp751.

On the other hand, for the multiplication on the Cortex-A family, the two-level Karatsuba Comba technique presented the best results, which comes in agreement with

Table 4. SIKE benchmarks on the Apple M1

		Operand Scanning (μs)	Reference (μs)			Operand Scanning (μs)	Reference (μs)
p434	**Key Generation**	1024	1316	**p503**	**Key Generation**	1725	1864
	Encapsulation	1673	2149		**Encapsulation**	2809	3073
	Decapsulation	1789	2297		**Decapsulation**	3008	3271
p610	**Key Generation**	3120	3576	**p751**	**Key Generation**	5526	5979
	Encapsulation	5783	6772		**Encapsulation**	8903	9693
	Decapsulation	5887	6653		**Decapsulation**	9606	10412

the approach taken in the reference SIKE ARMv8 implementation. For the multi-precision multiplication, the more aggressive instruction interleaving and use of the SIMD registers for intermediate storage presented small performance improvements of 2% or less.

For the Montgomery reduction in the Cortex-A family, we could observe consistent improvements with Operand Scanning in the in-order Cortex-A53 processor, with performance improvements ranging from 6% to 8%. Comparing the performance of the SIKE KEM implementations in the Apple M1, the Operand Scanning method provided performance improvements of 28% for SIKEp434, 8% for SIKEp503, 14% for SIKEp610 and 8% for SIKEp751 across the Key Generation, Encapsulation and Decapsulation procedures.

6. Conclusion

In this paper, we presented an evaluation of multi-precision multiplication techniques for the ARMv8 implementation of SIKE across its different security levels. We performed this evaluation on different ARMv8 platforms, including processors from the Cortex-A family and the Apple M1 SoC.

We observed that our implementation of the Operand Scanning technique for the multiplication resulted in significant performance improvements in the Apple M1 when compared to the reference implementation. For the Cortex-A family of processors, the Product Scanning (Comba) approach with two-level Karatsuba multiplication performed better. Our implementation of the two-level Karatsuba Comba, with more careful instruction interleaving and use of the SIMD registers as temporary storage for intermediate values, matched the reference implementation's performance on most cases. For the Montgomery reduction with Operand Scanning, we observed consistent improvements for the Apple M1 and Cortex-A53 when compared to the reference implementation.

From the results, we infer that the observed performance improvements could be originated from a better synergy of the straightforward Operand Scanning implementation with the Apple M1 out-of-order execution and its significant number of reorder buffers. We leave this observation as a future work and also propose evaluations of different approaches for multi-precision multiplication such as a hybrid between Operand and Product Scanning, as well as the evaluation of the Karatsuba multiplication with different levels and multiplication techniques. We also point to the development and evaluation of

different techniques that fully take advantage of the new ARMv9 architecture.

Acknowledgments

Part of results presented in this work was obtained through the "Post-Quantum Cryptography" project, funded by Samsung Eletrônica da Amazônia Ltda., under the Brazilian Informatics Law 8.248/91.

References

Castryck, W. and Decru, T. (2022). An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive.

Costello, C., Longa, P., and Naehrig, M. (2016). Efficient algorithms for supersingular isogeny Diffie-Hellman. In Annual International Cryptology Conference, pages 572–601. Springer.

Couveignes, J.-M. (2006). Hard homogeneous spaces. Cryptology ePrint Archive.

De Feo, L., Jao, D., and Plût, J. (2014). Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. Journal of Mathematical Cryptology, 8(3):209–247.

Faz-Hernández, A., López, J., Ochoa-Jiménez, E., and Rodríguez-Henríquez, F. (2017). A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. IEEE Transactions on Computers, 67(11):1622–1636.

Hofheinz, D., Hövelmanns, K., and Kiltz, E. (2017). A modular analysis of the Fujisaki-Okamoto transformation. In Theory of Cryptography Conference, pages 341–371. Springer.

Jalali, A., Azarderakhsh, R., Kermani, M. M., Campagna, M., and Jao, D. (2019). ARMv8 SIKE: Optimized supersingular isogeny key encapsulation on ARMv8 processors. IEEE Transactions on Circuits and Systems I: Regular Papers, 66(11):4209–4218.

Jalali, A., Azarderakhsh, R., Kermani, M. M., and Jao, D. (2017). Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM. IEEE Transactions on Dependable and Secure Computing, 16(5):902–912.

Koziel, B., Jalali, A., Azarderakhsh, R., Jao, D., and Mozaffari-Kermani, M. (2016). NEON-SIDH: Efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In International Conference on Cryptology and Network Security, pages 88–103. Springer.

NIST, N. (2017). Post-Quantum Cryptography. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.

Rostovtsev, A. and Stolbunov, A. (2006). Public-key cryptosystem based on isogenies. Cryptology ePrint Archive.

Seo, H., Sanal, P., Jalali, A., and Azarderakhsh, R. (2020). Optimized implementation of SIKE round 2 on 64-bit ARM Cortex-A processors. IEEE Transactions on Circuits and Systems I: Regular Papers, 67(8):2659–2671.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review, 41(2):303–332.