Fast Strategies for the Implementation of SIKE Round 3 on ARM Cortex-M4

Mila Anastasova[1], Reza Azarderakhsh[1] and Mehran Mozaffari Kermani[2]

[1]Florida Atlantic University, FL, USA
{manastasova2017, razarderakhsh}@fau.edu
[2]University of South Florida, FL, USA
mehran2@usf.edu

Abstract. The Supersingular Isogeny Key Encapsulation mechanism (SIKE) is the only post-quantum key encapsulation mechanism based on supersingular elliptic curves and isogenies between them. Despite the security of the protocol, unlike the rest of the NIST post-quantum algorithms, SIKE requires more number of clock cycles and hence does not provide competitive timing, energy and power consumption results. However, it is more attractive offering smallest public key sizes as well as ciphertext sizes, which taking into account the impact of the communication costs and storage of the keys could become as good fit for resource-constrained devices. In this work, we present the fastest practical implementation of SIKE, targeting the platform Cortex-M4 based on the ARMv7-M architecture. We performed our measurements on NIST recommended device based on STM32F407 microcontroller, for benchmarking the clock cycles, and on the target board Nucleo-F411RE, attached to X-NUCLEO-LPM01A (Power Shield), for measuring the power and energy consumption. The lower level finite field arithmetic and extension field operations play main role determining the efficiency of SIKE. Therefore, we mainly focus on those improvements and apply them to all NIST required security levels. Our SIKEp434 implementations for NIST security level 1 take about 850ms which is about 22.3% faster than the counterparts appeared in [SAJA20a]. Moreover, our implementations are 21.9%, 19.7% and 19.5% faster for SIKEp503, SIKEp610 and SIKEp751 in comparison to the previously reported results in [SAJA20a] for other NIST recommended security levels. Finally, we benchmark power and energy consumption and report the results for comparison.

Keywords: Supersingular Isogeny Key Encapsulation (SIKE), Post-Quantum Cryptography (PQC), Arm Cortex-M4

1 Introduction

The increasing capabilities of quantum computers are the motivation behind post-quantum cryptography (PQC) [Ber09]. Due to their data unit - the q-bit, and the principle of superposition, they are able to solve the hard mathematical problems underlying the classical cryptography in much shorter time than the nowadays computers. Shor's algorithm [Sho94] proves that factorization and elliptic curve discrete logarithm problems, the base of the widely used cryptosystems RSA and ECC, can be easily broken when quantum computers, equipped with enough q-bits, are developed. It is important to notice that the existence of such big machines is not yet achieved, however, once they are built, our private data, collected at any moment by a malicious party, could be easily decrypted in polynomial time, instead of exponential, when classical computers are used.

Due to the rising threat of quantum computers the National Institute of Standardization and Technology (NIST) [oSN] started a competition for post-quantum secure algorithms in 2016. Between the years 2017 and 2020 Round 1 and Round 2 of the competition were completed and their finalists - announced. The last round started in the year 2020 and still evaluates the candidates and their constant improvements, where the finalist algorithm is expected to be announce during the year 2021. The final round of the competition assesses two main groups - 9 Key Encapsulation Mechanisms (KEMs) and 6 Digital Signature Algorithms (DSAs). The main advantage of the supersingular elliptic curve-based cryptosystem, forming part of the alternate group of KEMs, is the compact size of the public keys and ciphertexts (i.e., 330 and 346 bytes for the NIST security level 1 implementation, respectively), which ensures insignificant communication latency. Taking into consideration the total timing - the computation cost and the data transmission, the size of the exchanged information results to be crucial, especially for the IoT and low-end real-time systems, where the traffic of data is enormous, and the fast information transmission is crucial for the functionality of the system.

The Supersingular Isogeny Key Encapsulation (SIKE) [JAC+17] scheme is based on the Supersingular Isogeny Diffie-Hellman (SIDH) algorithm proposed in 2011 by Jao and De Feo [PQC]. Both protocols rely on computations over elliptic curves similar, but more sophisticated, than the widely used Elliptic Curve Cryptography (ECC). Although several performance optimizations of ECC were proposed in the last years, targeting software and hardware [FA17, Seo20, NEKAMK20, NAK20], in the era of the quantum computers, this cryptosystem is not going to ensure securely transmitted information. To provide post-quantum resistance, SIDH and SIKE schemes are based on secret isogeny maps between supersingular elliptic curves, grouped in different isomorphic classes. These collections of curves are characterized by the $j-$invariant value of their elements, which is used as a unique identifier for each one of these classes. SIDH, however, shows vulnerability when one of the communication parties uses static key, allowing to break the algorithm in sub-exponential time [DF]. To eliminate this static key vulnerability SIKE was introduced as an IND-CCA algorithm, based on pseudorandom walks through supersingular elliptic curves.

Since the start of the NIST competition several research groups centered their work on the improvement of SIKE, aiming to reach efficiency in the computational time of the algorithm. In [SLLH18], [KJA+16], [SSJA20], [SAJA20b] the authors propose several strategies, targeting ARMv7-M, ARMv7-A and ARMv8 Arm based architectures, reporting significant speedup of the algorithm execution time. Several hardware implementations were proposed as well in [EAMK20a],[KAEK+20], [EAMK20b], [PLW+20] targeting the Xilinx Virtex 7 platform.

Contribution. In this work, we propose speed record results for the implementation of SIKE, targeting the resource restricted processor ARM Cortex-M4. Our contributions are itemized as follow:

- We propose to cleverly exploit the special form of the prime numbers used in SIKE for different NIST security levels and propose new techniques for reducing the number of memory accesses, therefore, decreasing significantly the clock cycles and the energy consumption of the protocol. Using these novel implementation strategies, we provide implementation speedup between 19.5% and 22.3% for all the NIST security levels of SIKE, where the maximum improvement is obtained for the prime SIKEp434 with 22.3% better performance, resulting in 1.28 times faster execution time.

- We propose a novel idea for the field multiplication, squaring and reduction operations. We use the floating-point register set as a level 1 cache memory, where we store the partial value of operations or the operand values. This allows us to significantly reduce the expensive memory access instructions. Moreover, it allows to increase the

row size of the multiplication, therefore, allows the implementation of completely different sequences of instructions for the reduction and squaring techniques.

- We propose efficient and optimized architectures for field modular addition. Our modular addition, based on continuous alternating block subtractions/additions, allows the implement the carry/borrow catcher, using one register for both, and allows us to perform less instruction for storing and activating the carry/borrow flag. This on the other side makes one register to be released, which permits to increase the size of the computational block. Moreover, based on the special form of the prime numbers, we decided to add the value of 2 to the modulus value, which results in several all-zero words and saves multiple subtraction instructions.

Organization. The rest of the paper is organized as follows. Section 2 reviews the mathematical background, needed for understanding of SIDH/SIKE. Section 3 shows the finite filed arithmetic operations underlying SIDH/SIKE protocols and the proposed optimization strategies, applied to the implementation. We describe in details the modular addition, multi-precision multiplication, squaring and Montgomery reduction. Section 4 shows the timing, power and energy consumption results. In Section 5 we summarize our work and the achieved results.

2 Preliminaries

This section presents an overview of the mathematical concepts, needed for understanding the supersingular isogeny elliptic curve-based protocols SIDH and SIKE. More detailed description of the algorithms is presented in documentation of SIKE [SIK].

2.1 Isogeny-Based Cryptography

The transition from classical cryptographic schemes to post-quantum algorithms will offer much higher security to the communication parties, due to the hard mathematical problems underlying the new encryption schemes. The mathematical concepts behind SIDH (base of SIKE) are founded on supersingular elliptic curves, isomorphic classes, j-invariants and isogeny maps.

A Montgomery-form elliptic curve is defined by the set of points satisfying the equation:

$$E_{a,b}/\mathbb{F}_{p^2} : by^2 = x^3 + ax^2 + x$$

together with the point at infinity, denoted by $\mathscr{O}$. In order to eliminate infinite sets, the elements, belonging to the curve, are defined over the quadratic extension of a finite field $\mathbb{F}_p$. The elements of the curve consist of two coordinates (x, y), where both coordinates are in $\mathbb{F}_{p^2}$, with $x, y = ai + b$ and $a, b \in \mathbb{F}_p$.

Unlike ECC, the SIDH scheme is based on pseudorandom isogeny walks through supersingular elliptic curves, defined over a finite field $\mathbb{F}_{p^2}$, which can be divided into roughly $\lfloor \frac{p}{12} \rfloor$ groups, called isomorphic classes. Each of them uniquely identified by the value of the $j-$invariant which is calculated as follows:

$$j(E_{a,b}/\mathbb{F}_{p^2}) = \frac{256(a^2-3)^3}{(a^2-4)}$$

The isomorphism is defined as a transition function between two elliptic curves, having the same $j-$invariant, which allows both communication parties to use its value as a shared secret for further efficient symmetric enciphering scheme, since they reach curves from the same isomorphic class.

Public Parameters: $p = \ell_A^{e_A}\ell_B^{e_B} \pm 1$, $E_0/\mathbb{F}_{p^2}$, $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$

Alice		**Bob**
1. $sk_A = m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$		1. $sk_B = m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$
2. $\phi_A : E_0 \to E_A$ with $ker(\phi_A) = \langle [m_A]P_A + [n_A]Q_A \rangle$		2. $\phi_B : E_0 \to E_B$with $ker(\phi_B) = \langle [m_B]P_B + [n_B]Q_B \rangle$
3. $pk_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$		3. $pk_A = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$
	$pk_A \longrightarrow$	
	$pk_B \longleftarrow$	
4. $\phi'_A : E_B \to E_{BA}$ with $ker(\phi_A) = \langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$		4. $\phi'_B : E_A \to E_{AB}$ with $ker(\phi_B) = \langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$
5. $ss = j(E_{BA})$		5. $ss = j(E_{AB})$

Figure 1: SIDH algorithm [PQC].

The pseudorandom walks among the curves consist of isogeny computations. Isogeny is more generic map between curves, which does not ensure the same $j-$invariant of the original and the image curves. By theorem of Tate [Tat66] two curves are isogenous only if they have the same number of points over a finite field. Velu's [Vél71] formula is used to find an isogeny ϕ and isogenous curve $E \to E/\langle G \rangle$, where G denotes the kernel of the isogeny and can be any finite subgroup from $E/\mathbb{F}_{p^2}$. The computational cost of the formula is high, especially for high degree isogenies. In SIKE, for performance optimization, the large isogenies are split into several maps of smaller degree, where isogenies of degree ℓ^e are split into e isogenies of degree ℓ. Therefore, Alice and Bob compute several "jumps" through the supersingular curves, forming pseudorandom walks on the isogeny graph.

The security of the isogeny-based cryptosystems relies on the difficulty to find the function, leading from E to $E/\langle G \rangle$ without knowing the kernel, whereas it is relatively simple to compute the isogeny map, knowing the subgroup. The best known attack to find an isogeny, knowing the original and image curves has a complexity of $O(\sqrt{p})$ and $O(\sqrt[3]{p})$, when using classical and quantum computers, respectively. While the security is significantly improved, since SIKE is believed to the quantum secure, its computational cost is much higher than the classical cryptosystems.

2.2 Supersingular Isogeny Diffie-Hellman

The Supersingular Isogeny Diffie-Hellman (SIDH) protocol is based on the idea of the classical Diffie-Hellman scheme, both of which aim to reach an agreement on a shared secret between two parties. Alice and Bob start from some public parameters, perform computations, based on their secret data, exchange the result that they have obtained, and in similar way apply again their secret key to the received data to finally reach a shared secret which allows them to start using much more efficient symmetric cryptosystems. While the idea behind both cryptosystem remains the same, the mathematical problem, underlying the schemes is completely different. SIDH is based on isogeny walks through supersingular elliptic curves, which increases the security of the system.

Figure 1 shows a high level overview of SIDH, where Alice and Bob start from a public elliptic curve $E_0/\mathbb{F}_{p^2}$, where p has the form of $\ell_A^{e_A}\ell_B^{e_B} f \pm 1$. In addition, both parties generate two basis points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$, respectively, and publish them. Alice uses her two points, together with her secret key, which consists of two integers $sk_A = m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, to compute her secret isogeny $\phi_A : E_0 \to E_A$ with kernel

$ker(\phi_A) = \langle [m_A]P_A + [n_A]Q_A \rangle$. Bob performs the same operations, where his secret isogeny is of the form $\phi_B : E_0 \rightarrow E_B$, with kernel of the isogeny $ker(\phi_B) = \langle [m_B]P_B + [n_B]Q_B \rangle$. After Alice and Bob reach the images of the public curve E_A and E_B, they find the projection of the other party basis point on their new curves. Finally, Alice and Bob form their public keys as $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ and $\{E_B, \phi_B(P_A), \phi_B(Q_A)\}$, respectively.

After both parties exchange this information they use, in a similar way, their secret key to reach curves, belonging to the same isomorphic classes, therefore, with the same $j-$invariant value. In particular, Alice will compute an isogeny $\phi'_A : E_B \rightarrow E_{BA} = E_B/\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$ and Bob - $\phi'_B : E_A \rightarrow E_{AB} = E_A/\langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$. Finally, they compute the $j-$invariant of the resulting curves as $j(E_{BA}) = j(E_{AB})$, using it as a shared secret for further communication.

The security of the SIDH algorithm is approximately $O(\sqrt[4]{p})$ and $O(\sqrt[6]{p})$ when using classical or quantum computers, respectively.

2.3 Supersingular Isogeny Key Encapsulation

SIKE was proposed in 2011 by Jao and De Feo [PQC] and was classified as an alternate candidate for the NIST Round 3 post-quantum standardization competition, where during the first two rounds it was optimized significantly and it is yet to be improved.

SIKE is a key encapsulation mechanism (KEM) where the parties of communication reach a shared secret, which is generated as a random message m and then encapsulated and decapsulated. Detailed graphical representation of the steps performed by both parties during the execution of the cryptography protocol is shown in Figure 2.

Figure 2 shows detailed description of the operations that Alice and Bob perform to reach a shared secret when using the post-quantum secure algorithm SIKE.

To perform the protocol Alice and Bob start, similar to SIDH, from a public supersingular elliptic curve $E_0/\mathbb{F}_{p^2}$, where the prime number p has the form of $\ell_A^{e_A}\ell_B^{e_B} \pm 1$ and the values of e_A and e_B vary depending on the NIST security level of the implementation. Both parties generate and publish two basis points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ that generate $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$, respectively. While performing the SIKE protocol, Bob generates his pair of public and private keys in the same way as in the SIDH algorithm, and sends his public key to Alice, whereas she is reusing his public key to encapsulate (using several hash functions denoted in Figure 2 as H, K and J) the random message value. In particular, Bob uses his secret key $sk_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$ and his base points to compute the kernel of his secret isogeny map $\phi_B : E_0 \rightarrow E_B = E_0/\langle P_B + [sk_B]Q_B \rangle$, which leads him to curve E_B. Similar to SIDH, Bob finds the projection of Alices' basis points and forms his public key, using the image curve together with the two projected points: $pk_B = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$.

In the scenario of SIKE Alice uses Bob's public key to perform two secret isogenies: she reaches curve E_A, using an isogeny map ϕ_A with kernel $\langle P_A + [r]Q_A \rangle$, where r is a hash of the random message m, concatenated with Bob's public key. Alice creates her public key as $pk_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$, using the new image curve and the projection of Bob's basis points on it. Moreover, she uses Bob's public key to compute a second isogeny, which starts from the image of Bob's curve using as kernel the projection of her basis points and the integer r : $\phi'_A : E_B \rightarrow E_{AB} = E_B/\langle \phi_B(P_A) + [r]\phi_B(Q_A) \rangle$. Alice generates the ciphertext that she sends to Bob as the concatenation of her public key and the XOR function of the secret message with a hash of the $j-$invariant $j(E_{AB})$. Finally, she computes the shared secret as a hash function of the concatenation of the random message and the ciphertext, that she has computed.

Bob, on the other side, tries to recover the value of the random message m that Alice masked in the ciphertext. He attempts to compute the value of the integer r and to reconstruct the same isogeny map, that Alice used to reach the second image curve E_{BA}. In particular, Bob computes an isogeny function $\phi'_B : E_A \rightarrow E_{AB} = E_A/\langle \phi_A(P_B) + [r]\phi_A(Q_B) \rangle$, which corresponds to the second isogeny performed by him in

Public Parameters: $p = \ell_A^{e_A}\ell_B^{e_B} - 1$, $E_0/\mathbb{F}_{p^2}$, $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$

Bob		**Alice**
1. $sk_B \in_R \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$ 2. $\phi_B : E_0 \to E_B$ with $ker(\phi_B) = \langle P_B + [sk_B]Q_B\rangle$ 3. $pk_B = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$ 4. $s \in_R \{0,1\}^t$	pk_B $\longrightarrow$	
		1. $m \in_R \{0,1\}^t$ 2. $r = H(m\|\|pk_B) mod \ell_A^{e_A}$ 3.$\phi_A : E_0 \to E_A$ with $ker(\phi_A) = \langle P_A + [r]Q_A\rangle$ 4. $pk_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ 5.$\phi'_A : E_B \to E_{BA}$ with $ker(\phi'_A) = \langle \phi_B(P_A) + [r]\phi_B(Q_A)\rangle$ 6. $c = (c_0, c_1) =$ $(pk_A, K(j(E_{BA})) \oplus m)$ 7. $ss = (J(m\|\|c))$
	$c = (c_0, c_1)$ $\longleftarrow$	
1. $\phi'_B : E_A \to E_{AB}$ with $ker(\phi'_B) =$ $\langle \phi_A(P_B) + [sk_B]\phi_A(Q_B)\rangle$ 2. $m' = c_1 \oplus K(j(E_{AB}))$ 3. $r' = H(m'\|\|pk_B) mod \ell_A^{e_A}$ 4. $\phi''_A :\ E_0 \to E_{A'}$ with $ker(\phi''_A) = \langle P_A + [r']Q_A\rangle$ 5. $pk'_A = \{E_{A'}, \phi''_A(P_B), \phi''_A(Q_B)\}$ 6. IF $pk'_A = c_0$ $ss = (J(m'\|\|c))$ ELSE $ss = (J(s\|\|c))$		

Figure 2: SIKE algorithm [JAC+17].

the SIDH protocol. Afterwards, Bob starts trying to recompute what Alice has already calculated: he finds the $j-$invariant of the curve E_{AB}, computes its hash function, which Alice used to mask the message m with, and reverses the XOR operation. He continues by recovering the value of r as the hash of m', concatenated with his public key, repeating the steps that Alice previously performed. He then computes the second isogeny map of Alice, using the value r': $\phi''_A :\ E_0 \to E'_A = E_0/\langle P_A + [r']Q_A\rangle$ and forms Alice's public key. In the final step Bob compares the public key of Alice from the ciphertext message and the one that he has obtained. In case they coincide, he calculates the shared secret in the same way as Alice did: $J(m'||c)$, where c is the ciphertext, received from Alice and J is hash function. In the case the value are different, he will use a random value s, created during the key generation phase, instead of the value of m' and will compute $J(s||c)$, which will prevent further communication between both parties.

During Round 2 of the NIST competition the parameters used for the implementation of SIKE were updated and Round 3 keeps these values. The public supersingular curve has the shape of $E_0/\mathbb{F}_{p^2} :\ y^2 = x^3 + 6x^2 + x$, with of $p = \ell_A^{e_A}\ell_B^{e_B} - 1$. The values for

Table 1: Summary of round 3 SIKE public parameters of SIKE submission [JAC+17].

Curve: $E_0/\mathbb{F}_{p^2}: \ y^2 = x^3 + 6x^2 + x$				
Parameter Set	NIST Security Level	Public Key Size (B)	Cipher Text Size (B)	Shared Secret Size (B)
SIKEp434	1	330	346	16
SIKEp503	2	378	402	24
SIKEp610	3	462	486	24
SIKEp751	5	564	596	32

$\ell_A = 2$ and $\ell_B = 3$ are fixed to ensure faster computations, however, the values of e_A and e_B vary, depending on the security that the implementation is covering. SIKE has four different primes which ensure NIST security level 1, 2, 3, and 5, respectively of length 434, 503, 610 and 751 bits. More detailed description of the length of the parameters of SIKE is presented on Table 1. The arithmetic computations over such big integers become challenging, especially when targeting resource restricted embedded devices. Therefore, the acceleration of such operations is the main objective for improving SIKE and its timing, power and energy consumption results.

3 Proposed Finite Filed Arithmetic Computations

The arithmetic operations, required for the execution of SIKE, are in a pyramid-like structure, consisting of several layers, where the highest one is formed by the SIKE complex isogeny computations, whereas the lowest one is composed by finite field arithmetic operations. The execution of high-level function requires several invocations of every consecutive lower level. Therefore, the speedup of the lowest level of the protocol computational pyramid ensures significant impact on the overall performance of the cryptosystem.

In this section we describe the modifications that we applied to the finite field operations to considerably decrease the number of clock cycles needed for the execution of SIKE.

3.1 Modular Filed Addition

Modular addition consists of adding two operands `A` and `B` and reduce the result modulo `p` in case it is not inside the finite field. This arithmetic operation requires one addition and one conditional subtraction. However, the development of a cryptosystem, that is secure against side-channel attacks, requires constant time execution of the operations independently of the operand values. For robust schemes, the modular addition is hence formed as one long-integer addition and one long-integer subtraction. The borrow produced by the subtraction determines the sign of the resulting number `T=A+B-p`, therefore, is used to form a mask which is `0xFFFFFFFF` when the result is negative and `0x0` when positive. Finally, the prime `p` is masked and added back to the result. In this way we obtain `A+B-p+p` if `A+B-p` is negative and `A+B-p` if `A+B-p+0` is positive. This implies two additions and one subtraction per operation. The strategy increases the timing of the finite filed computation, since it introduces a redundant addition, however, it ensures the constant time execution, eliminating the possibility of timing or power attacks.

Several works propose optimizations of this arithmetic operation. In [KPHS18] the authors provide an assembly implementation of the modular addition and subtraction operations. Further, in [SAJA20a] the authors introduce a new design, combining the first addition and subtraction, keeping the carry and borrow bits into general purpose registers (GPRs), exploiting a novel carry and borrow catcher technique.

In this work, we propose two novel ideas for outperforming the previous implementations

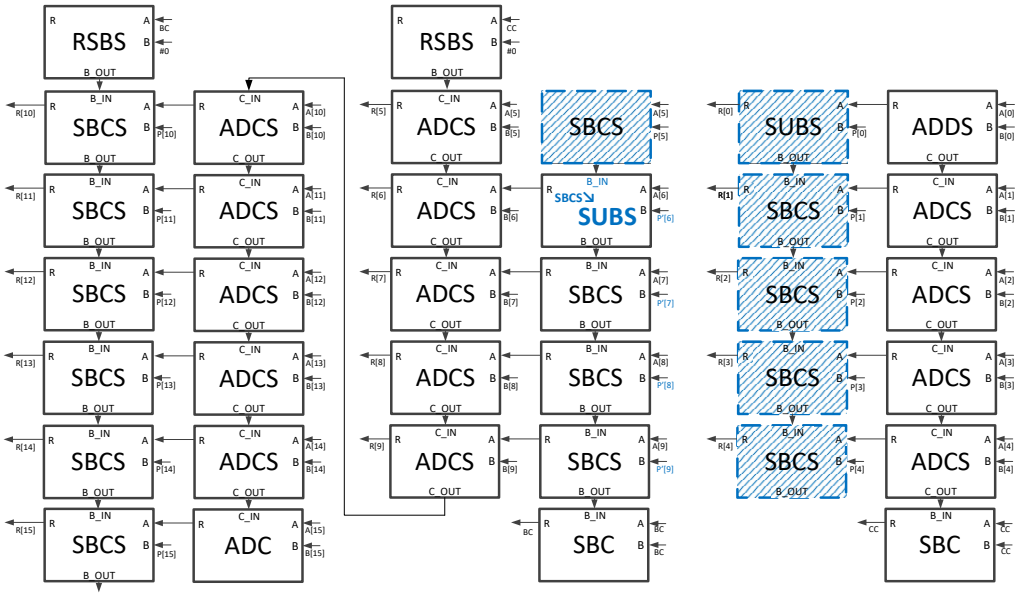


Figure 3: Proposed modular addition design with optimized Carry/Borrow Catcher (marked as CC and BC, respectively) due to the alternating sequence of addition and subtraction blocks. The first k subtractions (k depends on the prime number, used for the different NIST security level implementations of SIKE), marked in blue, are eliminated by replacing the value of `P`$= 2p$ with `P'`$= 2p + 2$.

of modular addition. First, by implementing an alternating sequence of additions and subtractions, our design requires less carry/borrow catcher operations. Moreover, by the "operation block flipping" strategy, we free one of the GPRs, which allows to execute more additions/subtractions at a time, where in the last block the number of limbs per operand may be increased to 6 as shown in Figure 3, since the registers, that keep the memory addresses of the operands, are freed. This reduces the number of non-consecutive memory accesses by using the `LDM` instruction for loading more 32−bit values from `A` and `B`. We free one of the registers, previously used to store the carry or borrow values, by the continuous switch between addition and subtraction blocks, as shown in Figure 3 (modular addition for p503), while the first set of instructions `(A+B)-P` is computed. Our design requires only one register to keep the carry or the borrow flag, since the operations are reversed, therefore, we store the value of the carry, activate the following carry flag when needed and then replace it by the borrow, produced by the last executed block.

On Figure 3, the carry flag is saved after the first 5 addition instructions, since the operand `A[0]-A[4]` is added with `B[0]-B[4]`. Later, the subtraction of the prime from the resulting value propagates the borrow using the `SBCS` instruction. The borrow flag is not saved after the first 5 subtractions, since afterwards we perform `A[5]-A[9] - P[5]-P[9]` and the borrow is propagated by the instruction itself. After 10 subtractions the carry flag is activated using the carry catcher register and then the borrow is stored in the same GPR. The carry/borrow catcher is implemented by subtracting a register from itself with the carry propagation option - `SBC`. We store the result inside the same register, calling it carry/borrow catcher register. Therefore, its value is `0xFFFFFFFF` in case the carry/borrow flag is active and `0x0` in case it is not. Later, when the next addition instruction is executed, the value of the flag is activated, depending on the carry catcher register value, using the instruction `RSBS`, which performs reversed subtraction. Using `0x0` as a second operand and the carry/borrow catcher register as the first, the flag will result activated if the catcher

register is equal to `0xFFFFFFFF` and will be deactivated in case it is `0x0`. The carry and the borrow catchers are implemented, using the same set of instructions - `SBC` to store the flag into the register and `RSBS` to activate the flag using the same register. The proposed operation block alternating strategy we reduce the number of carry/borrow store and activate instructions. Moreover, we free one register which is used to load more consecutive limbs from both operands into the register set. For instance, for p503, shown on Figure 3, we split the operands in only 3 subgroups instead of 4, as proposed in the previous best implementation design [SAJA20a], thus we reduce the number of non-consecutive memory accesses. Since the nature of the target platform allows memory access instruction of subsequent addresses to be executed in $N + 1$ clock cycles, where N is the number of 32−bit values loaded or stored, the reduction of the number of subgroups allows to decrease the number of clock cycles, produced by non-consecutive memory accesses, per modular addition operation.

It is important to note that, despite the alternation of the add/sub operations, the last performed instruction should be always a subtraction, since the borrow shows if the resulting number is positive or negative, thus determines the mask value for the final addition with `P`. Therefore, the first operation block (addition or subtraction) depends on the prime number used for SIKE security level 1, 2, 3 and 5, such that the final borrow can be kept in the carry/borrow flag.

In this work we also propose to use the special form of the SIKE primes to optimize even more the quantum secure scheme. As seen before, p has the form of $2^{e_A}3^{e_B} - 1$ which, due to the value of the exponents, results in a number with several 32-bit all-one words at the end of the prime. The reduction inside the modular addition of SIKE uses the value of `P`$= 2p$, where the $\times 2$ operation simply results in shifting the number 1 bit to the left. Therefore, the last words of `P` consist of several $1's$ and a 0 at the end. We noticed that if we add 2 to the value of `P`, `P'=P+2`, all the $1's$ followed by a 0 at the end are be replaced by all-zero words, where the first non-zero digit is increase by one. Based on this idea, we propose an implementation of the modular addition as follows:

```
(A+B)-(2P+2)+((MASK2P)+2)
```

The proposed idea is graphically represented in Figure 3. Since the last k words of `P'` are all-zero, we can skip as many subtractions as number of words equal to 0. The value of k depends on the SIKE prime value. Since we subtract `2P+2` from the partial result, we add again 2 to the masked value of `2P`. In this case we add back the value of 2 as a final addition or the value of `2P+2`. The eliminated subtraction instructions are shown in blue on Figure 3. This implementation design results in significant speedup due the considerable reduction of instructions.

Our two novel proposals for the implementation of modular addition result in outperformance of the previous development designs and significantly improve the overall timing results for the post-quantum secure algorithm. Moreover, since the proposed implementation is completely scalable, it is adapted to all the SIKE primes.

3.2 Field Multiplication, squaring, and reduction

The modular multiplication, used for the implementation of SIKE, is Montgomery multiplication, which performs multiplications and reduction. It consists of two multiplications and an addition, where the specific form of the prime numbers, used for the post-quantum secure protocol, allows several techniques for reducing the execution time. In this section we propose novel idea for reducing the number of clock cycles of the multiplication, the reduction and the squaring functions. The scalability of our design allows the adaption of the implementation to different lengths; therefore, we implemented all the four different secure levels of SIKE and observed a significant speedup for all of them.

The architecture of the target platform of this work - Arm Cortex-M4 - promises 1 clock cycle (CC) per instruction, except for memory access operations, which require 2 CCs. Therefore, the implementation designs, based on this processor, are focused on reducing the number of load and store instructions. We present a novel implementation strategy, which minimizes the memory access instructions. We propose to use the entire set of floating-point registers (FPRs) simulating level 1 cache memory, requiring only 1 CC per instruction. Therefore, instead of accessing the memory, which requires 2 CCs, we move the values from general purpose registers (GPRs) to FPRs. The `VMOV` instruction requires only 1CC independently of the order of instructions and ensure that no data dependencies or structural hazards are produced. This novel idea is the base of our implementation designs for the multiplication, reduction and squaring operations. Table 2 shows detailed number of the memory instructions used in several implementations, including our work, where we replace some of the memory accesses by moving instruction between the two types of register sets.

3.2.1 Field Multiplication

The multi-precision multiplication operation forms large part of the cryptographic protocols since the computationally expensive operations (i.e., inversion) are replaced by several multiplication invocations. This motivates several research groups to focus on the performance of this operation. The implementation shown in [KPHS18] is based on the Karatsuba multiplication [KO62], where instead of performing one multiplication of operand sizes n, three half-size multiplications are completed together with several additions/subtractions. The time complexity of the Karatsuba algorithm is $\mathcal{O}(n^{log_2 3})$, whereas the Product Scanning (PS) approach has time complexity of $\mathcal{O}(n^2)$, therefore, the proposed implementation shows better performance. Furthermore, in [DSS16] the authors implement a 2-level Karatsuba multiplication ending with several 64×64-bit multiplications instead of one 256×256-bit operation. However, due to the architecture of the target processor, the use of the low cost Multiply ACcumulate (MAC) instructions is proposed in [HW11] and is integrated in the big-integer multiplication design, since no stall are introduced after the complex instruction execution. The use of MAC instructions, combined with reduced number of memory accesses, leads to the most efficient implementation algorithms for the target platform Arm Cortex-M4, named Operand Caching (OC) [HW11] and its variants. As the name stands, it is focused on the reuse of operands once they are loaded into the register set. This strategy reduces the load and store instructions by introducing the concept of rows. In each row several values of the operands `A` and `B` are cached in the GPR set. The size of the row is defined as the number of accumulative multiplications performed per column. The OC approach acts as Product Scanning (PS) multiplication inside each row and as Operand Scanning (OS) between the rows, where the value of the partially accumulated result is stored into the memory and loaded later when the following row is being computed. Further improvements of the algorithm are presented by Seo et *al*, in [SK12] and [SK15], where the Consecutive Operand Caching and the Full Operand Caching implementations are proposed, aiming to further optimize the memory accesses by reconfiguring the sequence of executed instructions .

In [FA17] the use of the MAC instruction `UMAAL` is evaluated aiming to eliminate the need of carry bit propagation through the limbs of the partial result value. The MAC instruction consists of one $64 \leftarrow 32 \times 32$-bit multiplication accumulated with 2 32-bit values and handles the additions without producing an overflow, therefore no carry propagation is required using the carry flag. The set on MAC instruction is also considered in [HL19], where the combination of `UMLAL` and `UMAAL` instructions is presented. However, the use of `UMLAL` requires the initialization of the register that keeps the high 32 bits, which introduces one additional clock cycle. In [SJA19] the authors propose optimized strategy, integrating the instruction `UMULL`, which handles the initialization of the register while the 32×32-bit

Table 2: Memory accesses and GPR to FPR move instructions count. For the multiple load instructions, the number of registers loaded is shown inside brackets. We measure the number of clock cycles needed to perform all the instructions to compare the different implementation strategies.

Memory accesses									
Design	SIKEp434				SIKEp503				
	LDR	STR	VMOV	Total [CC]	LDR	STR	VLDM	VMOV	Total [CC]
OS	406	210	-	1232	528	272	-	-	1600
OC	132	80	-	424	172	102	-	-	548
R-OC	107	63	-	340	140	80	-	-	440
This work	70	0	100	240	24	34	2(×16)	146	296
Design	SIKEp610				SIKEp751				
	LDR	STR	VMOV	Total [CC]	LDR	STR	VLDM	VMOV	Total [CC]
OS	820	420	-	2480	1176	600	-	-	3552
OC	268	154	-	844	384	216	-	-	1200
R-OC	215	120	-	670	306	168	-	-	948
This work	131	12	188	474	198	32	-	240	700

accumulative multiplication is performed. Later, they improve the design even more in [SAJA20a], where they implement efficient multiplication strategy for all SIKE primes. They propose novel management of the register set for caching four words per operand, therefore, they increase the size of the rows in comparison to the previous implementations.

In this paper, we base our work on [SAJA20b, SAJA20a], where we propose new ideas to increase even further the size of the rows, if needed. Depending on the form of the prime, we implement two different strategies, both based on the FPRs use. For the first implementation strategy, we use the FPR set to load the values of the operands at the beginning of the algorithm so that we do no need to access the memory to load them into the GPR set. For the second implementation, we replace the usage of the stack with the FPR set, using it as a level 1 cache memory for the partial results of the multiplication. Both implementation strategies improve significantly the performance and allow even further optimization of the design.

For representing the multiplication we use rhombus notation, where each diagonal line shows a 32-bit limb from the operand `A` or `B`. The limbs from both operands are shown as `A[` k `], B[` k `]`, where $k \in \{n-1, ..., 2, 1, 0\}$ with 0 being the least significant 32-bit word. The number of limbs n varies, based on the number of bits needed to represent the integer m and the processor word size w, thus $n = \lfloor m/w \rfloor$. The number of words needed for the multiplication result is double, where `R=(R[` $2n-1$ `],...,R[ 1 ],R[ 0 ]) =` $\mathtt{A} \cdot \mathtt{B}$. In the multiplication rhombus notation every dot shows a 32×32-bit multiplication , where the operands are the two 32-bit limbs, represented by the crossing diagonals. Finally, the bold vertical lines show the addition of all the partial 32×32-bit multiplication products. We also use box representation to show the sequence of multiplications and additions in more details.

This work proposes two different approaches for the use of the FPR set. Based on the length of the operands, we propose distinct multi-precision multiplication optimization strategies. In the 256-bit examples in Figure 4 and Figure 5 we show both strategies that are used in our implementation design. On Figure 4, we propose to increase the size of the

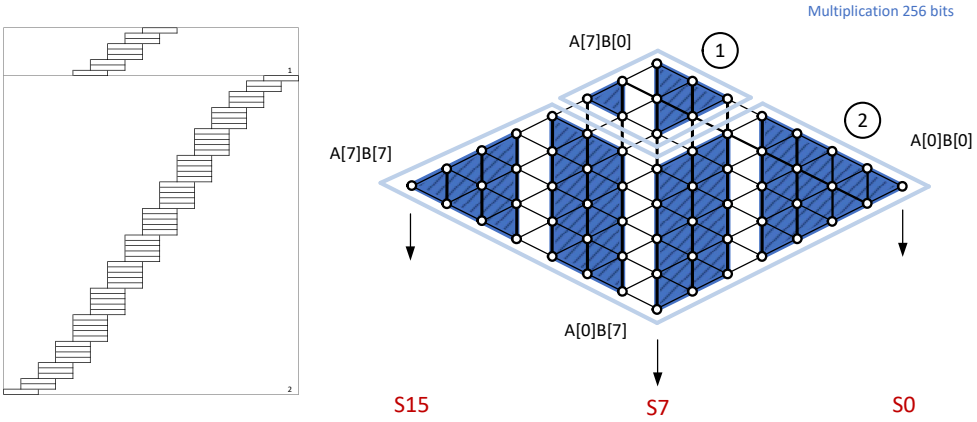


Figure 4: Rhombus representation of 256 × 256-bit multiplication, presenting the increased row size. The technique is implemented by using the FPR set to load the values of the operands at the beginning of the multiplication, which allows load of the operand limbs in 1 CC.

rows by performing 5 accumulative multiplications. We do that by reserving 5 registers for the operand `A` and 6 registers for the resulting values. The 3 free registers left we use for the second operand `B`. Since we need 5 words from `B`, we need to re-load their values constantly. However, we propose to use the FPR set to store the value of the operands. We perform a multiple vector load at the beginning of the algorithm. Later, instead of loading the values of the operands from the memory, requiring 2 CCs, we move them from FPRs to GPRs, requiring only 1 CC. The computation of each 4 × 4-column requires the load of a new limb of `B`. Instead, our design requires to load one new word and to re-load a previous one. However, we replace the load instruction by `VMOV` and therefore use the same amount of clock cycles for the computation of the row marked with **2** in Figure 4. The speedup in our design is introduced, therefore, by the reduction of the previous row length, which requires less load and store operations for the partial result. Depending on the prime, the increased size of the rows may result in less number of rows, therefore multiple memory access instructions are eliminated.

The next multi-precision multiplication strategy is shown on Figure 5. In this design, which is used for p434, p610 and p751 we do not modify the size of the rows. We use the FPRs as a cache memory to store the partial values, produced by the rows. Depending on the resulting size, where the number of FPRs is not enough to store the final result, the stack is used to store the last 8 and 16 limbs, for p610 and p751, respectively.

Following are presented some specifications about the implementation strategies, depending on the prime numbers.

SIKEp434 The length of the prime requires $n = \lfloor m/w \rfloor = \lfloor 434/32 \rfloor = 14$ words to store each operand. In [SAJA20a], the multiplication for p434 is divided into four rows, shown in Figure 6, where the initialization row, marked with **1** in the Figure 6, produces only 4 words as a partial result. Therefore, the idea of increasing the size of the rows and decreasing their number could not be efficiently applied. The reduction of rows will result in only 4 less instructions for storing the temporary value from row **1** and another 4 instructions for loading the words, when accumulating them with the partial result from row **2**.

Our proposal for the prime p434 consists of replacing the usage of the stack for the partial result with FPRs. The temporary values produced by each row are moved into the

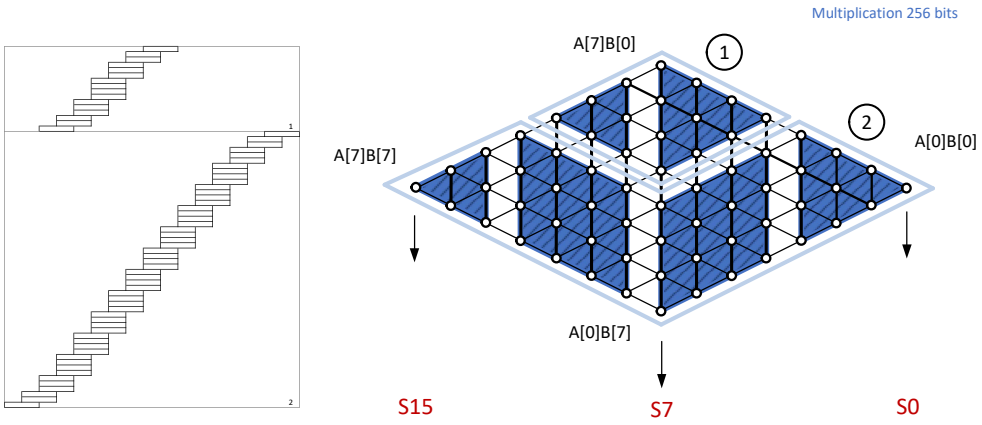


Figure 5: Rhombus representation of 256 × 256-bit multiplication, using the FPRs as a cache memory level 1 for the partially computed results from each row.

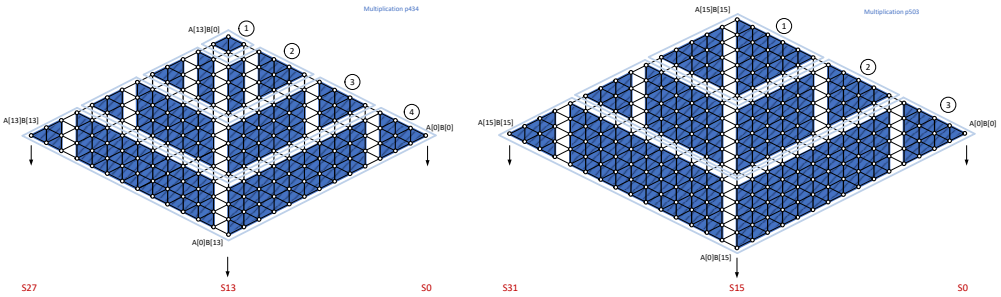


Figure 6: Rhombus representation of 448-bit (used in p434) and 512-bit multiplication (used in p503). The smaller multiplier uses the FPR set as a cache memory for the partial results, avoiding using the stack. The 512-bit multiplier uses the FPR set to store the operands and hence to reduce the cost of accesses to their limbs, resulting in reduction of the number of rows.

FPR set, where the cost of the `VMOV` instruction is only 1 CC in comparison to the `STR` and `LDR` instructions, which double the required cycles.

The multi-precision multiplication of two 14-word operands produces 28-word value. We noticed that there are another 4 FPRs which remain free. We have proposed to store the memory address of the operand `A` and `B` into 2 of these registers. Due to the reduced number of GPRs, the pointer to `A` and `B` are not kept into the core register set, therefore, they should be constantly loaded. The use of the FPRs, storing the addresses, optimizes even further the memory accesses and results in even more efficient multiplication algorithm.

SIKEp503 The prime number, consisting of 503 bits, requires 512 bits to store the value of the operands, where the formula $n = \lfloor m/w \rfloor$ indicates the need of 16 words to keep them. The number of limbs for both operands `A` and `B` could fit in 32 $32 - bit$ registers. We have noticed that this value is the same as the number of FPRs, where we decided to load them. This allows the elimination of all memory accesses for loading the operands, replacing them by the `VMOV` instruction.

At the beginning of the algorithm, we load the value of the operands into the FPR

Algorithm 1 Management of the GPRs when the `VMOV` instruction is used for the increased row size in the case of 5 consecutive accumulative multiplications.

```
VLDM R0, {S0-S15}
VLDM R1, {S16-S31}
....
UMAAL R0, R14, R1, R6
UMAAL R0, R12, R2, R7
UMAAL R0, R11, R3, R8
VMOV R6, S6
UMAAL R0, R10, R4, R6
VMOV R6, S10
UMAAL R0, R9, R5, R6
STR R0, [SP, 4*16]
LDR R0, [SP, 4*17]
UMAAL R0, R14, R1, R7
UMAAL R0, R12, R2, R8
UMAAL R0, R11, R3, R6
VMOV R7, S7
UMAAL R0, R10, R4, R7
VMOV R7, S11
UMAAL R0, R9, R5, R7
....
```

set, using the `VLDM` instruction, which requires $k + 1$ clock cycles, with k representing the number of single precision FPRs. Therefore, each operand will require $n + 1$ cycles to load it into the register set. The big integer multiplication, implemented on resource restricted devices, does not allow to store the value of the limbs all at once in the register set. Even more, the GPR set of the processor Arm Cortex-M4 offers only 14 core registers, which can be used to keep user data. Therefore, a constant reload of the operand words is needed. We have performed the implementation of p503, based on the idea of loading the values of the operands in the FPR set, and thus reuse them in much more efficient way.

Algorithm 1 shows how we manage the usage of the GPRs. After loading five consecutive limbs from `A`, each following column computation requires to load one new limb of `B` and to reload a previous one. This imposes a constant 2 CCs per column for loading 2 operand words, when using the `VMOV` instruction. Therefore, the 5×5-limb multiplication has the same cost as the 4×4 multiplication, proposed in [SAJA20b], where the `LDR` instruction is used.

The optimization of the implementation, resulting in better execution time, is due to the increase of the row sizes, which leads to decrease of the number of rows, shown on Figure 6. Since the design stores the partial results in the stack, the reduction of rows significantly cuts the number of memory accesses, since less partial results values should be stored and loaded. We increase the size of the rows from 4 to 5, which reduces the number of rows and results in implementation, outperforming significantly the previous designs.

SIKEp610 and SIKEp751 The length of the primes requires $n = 20$ and $n = 24$ words, for p610 and p751, respectively. The large size of the operands does not allow to store them into the FPR set like the p503 implementation, since we are not able to eliminate all `LDR` instructions that access the operands. For the large primes p610 and p751 we propose to use the FPR set to store the partial result, formed after computing each row, similar to the p434 implementation, which reduces the stack usage. However, the resulting values, consisting of 40 and 48 words respectively cannot fit into the FPR set entirely. We have developed a solution that uses the FPRs for the storage of the partial results of the first

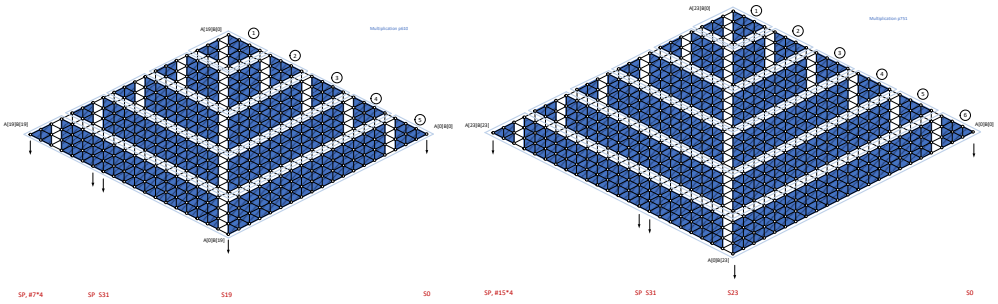


Figure 7: Rhombus representation of 640-bit and 768-bit multiplication (used in p610 and p751, respectively) based on the implementation of reduced (but not eliminated) stack usage by storing the partial values in the FPRs as a level 1 cache memory.

four rows for both primes and then uses the stack for the last 8 or 16 words, computed in the last row/s, respectively, for p610 and p751, as presented on Figure 7.

For further improvement, we store the least significant 32 words of the result into the FPRs and most significant $n-32$ words into the stack, where $n = \lfloor m/w \rfloor$, which optimizes the stack usage, since the following reduction uses the least significant n words as an operand for the second multiplication operation and should be loaded much more often, while the n most significant words are used for the accumulation only, therefore are loaded less times.

3.2.2 Modular Reduction

The post-quantum secure protocol uses Montgomery multiplication, since the reduction step takes advantage of the special form of the "Montgomery-friendly" prime numbers, used for the four different NIST security levels. The implementation of this operation is impacted by the techniques used in the design of the multi-precision multiplication. The reduction implementation requires another multiplication together with an addition operation. The "Montgomery-friendly" form of the primes ensures that the least significant k operations from the multiplication are skipped, since they consist of $\times 0$ multiplications. More specifically, for NIST security level 1, 2, 3, and 5 $k = \{6, 7, 9, 11\}$, respectively. The optimization technique was proposed by Costello et al. in [CLN16], pointing out the advantages of Montgomery reduction for the implementation of SIKE due to the special form of prime, where the previously used Barret reduction was replaced by Montgomery multiplication in the new implementation design.

The Montgomery reduction is crucial for the efficient implementation of the isogeny-based quantum secure algorithm. Several works have studied the implementation strategies suitable for the design of the arithmetic operation. In [LG14], an optimized Hybrid Montgomery multiplication method was proposed and evaluated, targeting small 8-bit AVR microcontrollers, showing significant improvement in the timing results. In [SLLH18], Seo at al. proposed even further acceleration of the Hybrid Scanning method, benefiting from the MAC instructions in the SISD implementation of the algorithm. In [SAJA20b] the authors propose several optimizations for the implementation of the multiplication, increasing the row sizes to 4, and apply them to the reduction, where they show a performance record of the arithmetic operations.

In this work, we propose novel implementation of the reduction algorithm, which optimizes the number of rows and therefore reduces the number of memory accesses significantly. Our reorganization of the sequence of instructions, along with the usage of the FPR set, presents significant optimization of the algorithm performance in comparison

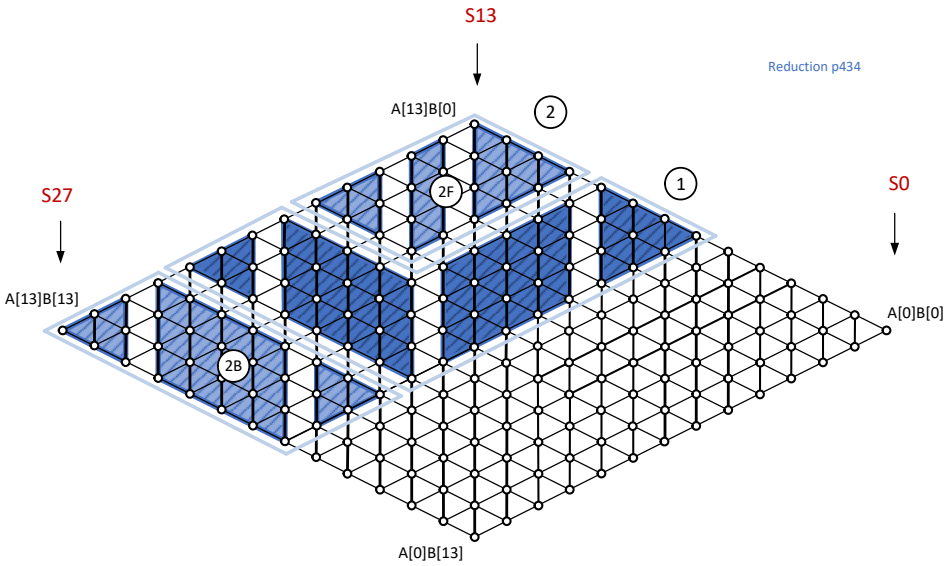


Figure 8: Reduction for p434, presenting the novel shape of the rows. The order of the rows is marked in circles, where **2F** and **2B** denote the front and the back part of the second row.

to the previous best implementation.

We apply the optimization strategy of the increased row size, if needed, as described in the multiplication implementation - Section 3.2.1. Depending on the prime value and the number of all-zero words at the least significant positions, we have decreased the number of rows with 2 ($4 \rightarrow 2$), 2 ($4 \rightarrow 2$), 2 ($5 \rightarrow 3$) and 3 ($6 \rightarrow 3$), respectively for p434, p503, p610 and p751. For NIST security level 1 and 3 primes we have kept the row size to 4 (where for p610 we observed more efficient implementation when we decrease one of the row sizes to 3), however, for NIST level 2 and 5 we have modified them as presented on Figure 8 and Figure 9.

The optimization of the number of rows is implemented applying a novel strategy for reduction shown in Figure 8. The multiplication sequence follows completely different direction, compared to the previous implementation designs. In this work, we approach the multiplication, starting from the least significant non-zero values of M and multiply them with the values of Q, where Q denotes the least significant n words from the result $T = A \cdot B$. The Montgomery reduction requires the accumulation of T with the resulting value from $M \cdot Q$, before a word from Q can be used for following multiplications. Therefore, we use the values of $Q[0]$-$Q[k]$, without accumulating them since M ends with all-zero words, where k is the index of the most significant all-zero word of M. The first m computed words of the reduction, where m is the size of the first row, accumulated with the previous value of $T[k+1]$-$T[k+m]$, allow the use of $Q[0]$-$Q[k+m]$ during the computation of the first row, since the value is complete and no further accumulation is needed. Therefore, we extend the length of the first row until the limb $k+m$ from the accumulated operand Q is needed. Afterwards, we reuse the last m values of Q and load one word from the operand M per computed column. The implementation technique requires that the direction of the multiplications is changed. Specifically, we load the values of the first m words from M and reuse them, while loading the values of Q. Once we reach the $Q[k+m]$ limb (while we perform the first row of the multiplication), we change the direction of the operation, where we reuse the cached values of the accumulated Q in the register set and start loading the values of the operand M. The following rows implement the same idea,

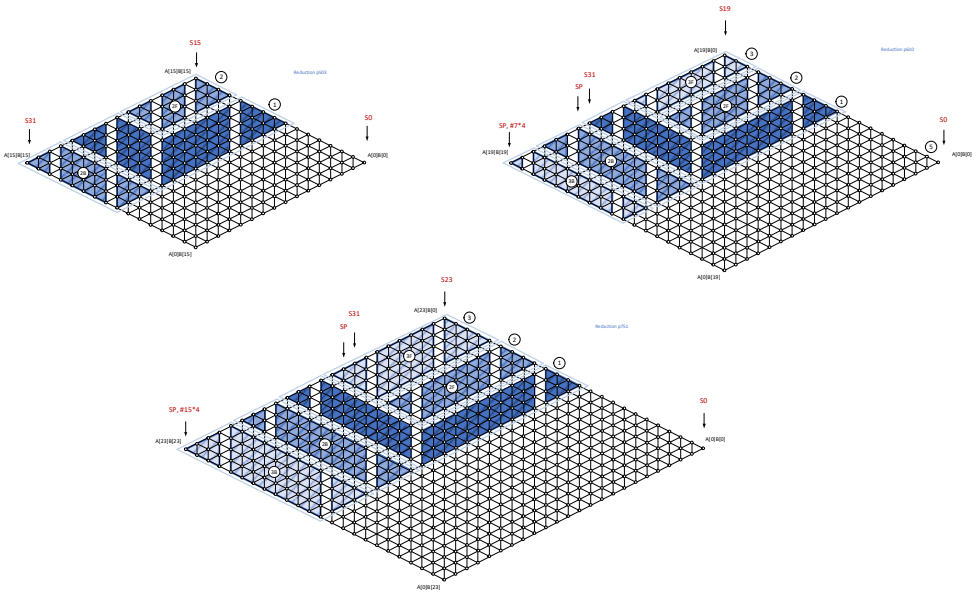


Figure 9: Reduction for the primes p503, p610 and p751. The front part of a row calculation is denoted by nF and with nB, where n is the row number.

where although it appears that the rows are interrupted in the middle, the same row size is conserved. Therefore, the computation of each column through the rows consists of 4 (or 5) accumulative 32×32-bit multiplications. We mark these rows as $n\mathbf{F}$ and $n\mathbf{B}$ on Figure 8 and Figure9, where n is the number of the row. This technique requires to load cartain limbs of M several times. For instance, on Figure 8, the second row requires $M[10]$-$M[13]$ during the computation of **2F**. When **2B** is being calculated the value of $M[6]$-$M[9]$ is needed at the beginning of the zone, where afterwards the values of $M[10]$-$M[13]$ should be reloaded. However, despite the reload of particular words of M, the reduced number of rows, together with the usage of the FPRs, keeping the value of the operand Q and later used as a storage for the partial results of the multiplication, results in significant improvement, where the reloading instructions end up negligible for the timing results.

As mentioned, some of the primes require row size of 5, therefore, for further improvement of these rows we keep 5 words from the operand M in the registers, since it requires 2 clock cycles to move it into the GPRs, using `MOVW` and `MOVT`, and only 3 words of the operand Q, where we need only 1 clock cycle to move their value from the FPR set to the GPR set.

The implementation of the Montgomery reduction, as mentioned, requires the accumulation of the $Q \cdot M$ multiplication result to the value of the temporary $T = A \cdot B$ multiplication. The value of Q is the least significant n words from T, therefore the value of the low part of T is accessed significantly more times than its high part, which is required only for accumulating the result. To increase the performance result for the integers that do not fit in the FPR set only, we have placed the least significant value of $A \cdot B$ using the FPRs which ensures one clock cycle per `VMOV` into the GPRs, whereas we have placed the extra 8 or 16 words for p610 and p751, respectively in the stack, due to their low access rate.

We present all the reduction implementation strategies on Figure 9, where we show in detail the size of each of the rows and the number of rows needed for each prime number.

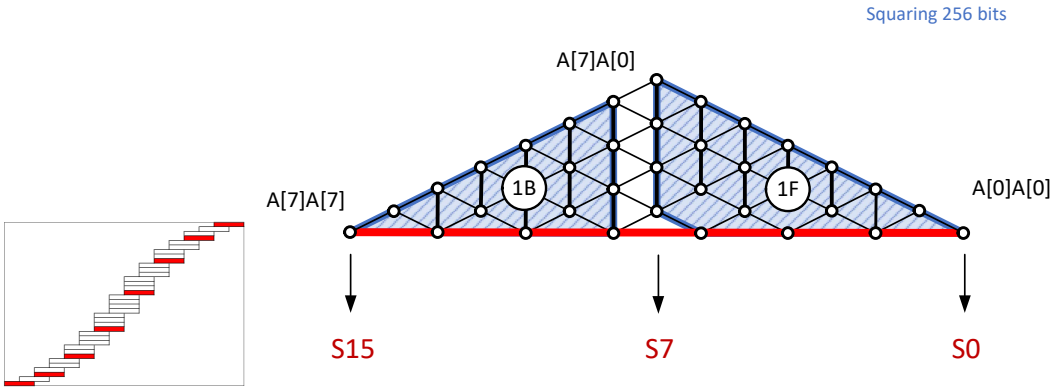


Figure 10: 256-bit squaring, where the rows are denotes in circles. The small size of the operands allows to show only the sub-squaring type of rows, used in the implementation. 1F denotes the form part and 1B the back part of the operation.

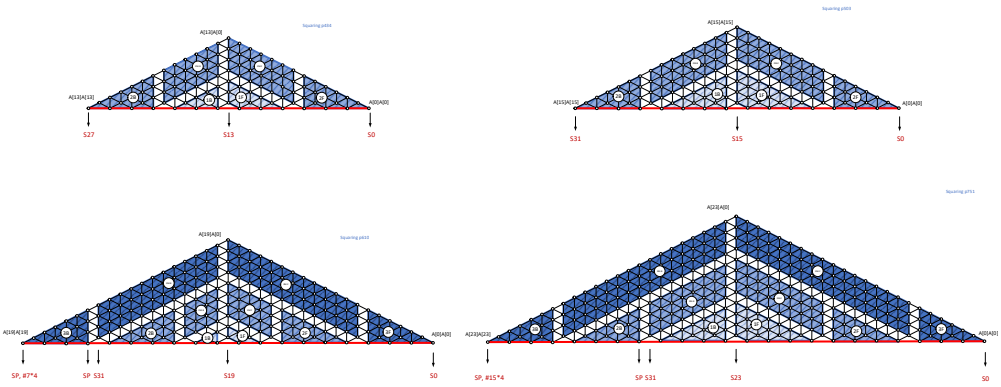


Figure 11: Squaring implementations for all the SIKE primes. The sub-squaring blocks are denoted with n**F** and n**B** and the sub-multiplication block with n**M-F** and n**M-B** (representing **M**iddle **F**ront and **B**ack).

3.2.3 Squaring

Squaring is a special case of multiplication where the operand A and B are the same, therefore, for the implementation of this arithmetic operation several further optimizations can be applied. Since the two operands have the same value the number of memory access instructions can be significantly reduced if the limbs are properly reused. The rhombus representation of the multiplication can be split into three parts: upper part, where the operands of the multiplication are different (i.e $A[i]A[j]$), middle part, where the operands of the multiplications are the same (i.e $A[i]A[i]$), and bottom part, which produces the same results as the upper part with reverse indexes (i.e $A[j]A[i]$). Therefore, while performing the squaring, the computation of the bottom part can be eliminated by doubling the result of the upper part. There are two strategies for the $\times 2$ multiplication of this part, where one is based on the doubling of the result and the other - on doubling of one of the operands. In [HL19] the authors use the former implementation, whereas Seo at *al.* developed the later strategy in [SAJA20b]. In our proposed implementation we are using the later design, where we compute the value of $2 \times A[i]$ and then multiply by $A[j]$ to obtain $2 \times A[i]A[j]$.

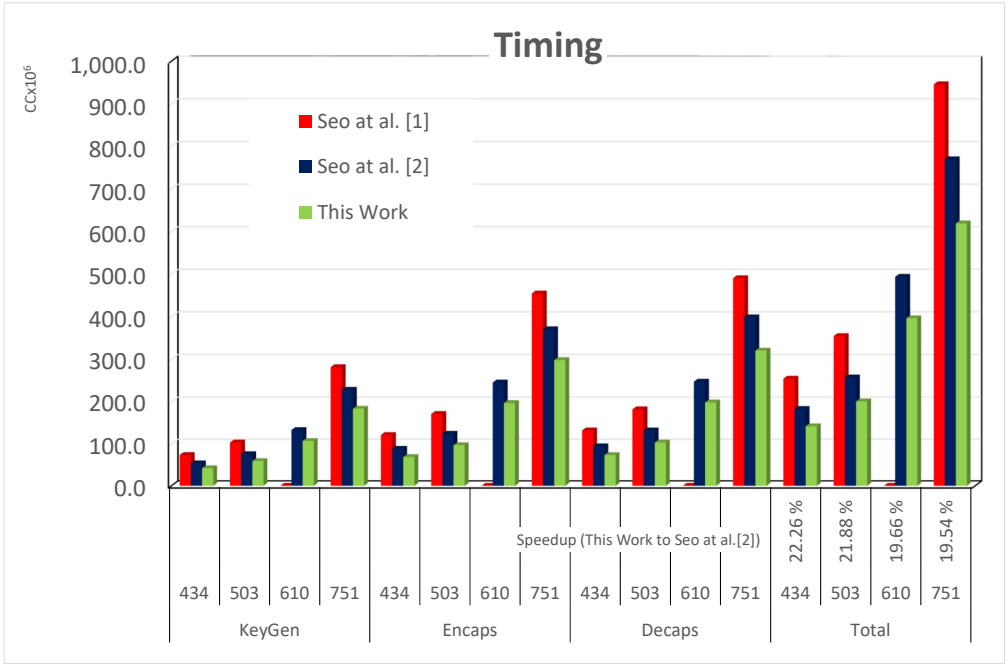
Table 3: Comparison between the SIKE implementation targeting Cortex-M4 and our new implementation design. Results Clock cycles and the seconds on STM32F407 of SIKE with the proposed optimizations integrated in the implementation. The timing results are presented in clock cycles and in seconds. The speedup obtained is marked in red color.

Implementation	Language	Timing [cc×10^6]				Timing[s]				Speedup
		KeyGen	Encaps	Decaps	Total	KeyGen	Encaps	Decaps	Total	[%]
SIKEp434										
SIDH v3.3 (SIKE R3)[1]	C	718	1,175	1,254	2429	4,27	6,99	7,46	14,46	94.1
Seo et al.[2]	ASM	74	122	133	255	0.44	0.73	0.79	1.52	43.91
Seo et al.[3]	ASM	54	89	95	184	0.32	0.53	0.57	1.09	22.5
This work	ASM	42	69	74	143	0.25	0.41	0.44	0.85	-
SIKEp503										
SIDH v3.3 (SIKE R3)[1]	C	1076	1773	1886	3659	6.40	10.55	11.23	21.78	94.5
Seo et al.[3]	ASM	104	172	183	355	0.62	1.02	1.09	2.11	43.2
Seo et al.[2]	ASM	76	125	133	258	0.45	0.74	0.79	1.54	21.6
This work	ASM	59	97	104	201	0.35	0.58	0.62	1.20	-
SIKEp610										
SIDH v3.3 (SIKE R3)[1]	C	2011	3701	3722	7423	11.97	22.03	22.16	44.18	94.5
Seo et al.[3]	ASM	134	246	248	494	0.80	1.46	1.48	2.94	17.5
This work	ASM	108	198	199	397	0.64	1.18	1.19	2.43	-
SIKEp751										
SIDH v3.3 (SIKE R3)[1]	C	3647	5915	6353	12267	21.71	35.21	37.81	73.02	94.9
Seo et al.[2]	282	455	491	946	1.68	2.71	2.92	5.63		34.5
Seo et al.[3]	ASM	229	371	399	770	1.36	2.21	2.38	4.58	19.5
This work	ASM	184	299	321	619	1.10	1.78	1.91	3.69	-

The referred results are presented in:
[1] [SIK]
[2] [SJA19]
[3] [SAJA20b, SAJA20a]

The previous implementations of the modular squaring include the Sliding Block Doubling (SBD) with Bottom Line and Initial Block algorithm [FA17], where the authors introduce the use of initial block for optimizing the register. Later, implementation of the squaring, based on the Operand Scanning strategy, was proposed in [HL19]. However, the proposed design is particular for integers of length 256 bits, therefore, it cannot be adopted to other operand sizes. In [SAJA20b] the authors propose an efficient implementation, where the operation is split into several parts. For 256-bit length of the operands, the authors define 3 rows. One row is defined as sub-multiplication type, following their Refined-Operand Caching multiplication optimizations. The other 2 zones are defined as sub-squaring type, where they use the SBD technique. Moreover, they precompute the double of the operand and used the stack to save the values, therefore, they apply the double-operand technique for the $\times 2$ computation.

In this work, we propose new implementation strategy for the modular squaring which reorders the instructions to optimize the memory accesses. Our design, similar to our proposed multiplication, uses the FPR set to store the partial values after the computation of each row. Moreover, due to the absence of second operand, there are several unused FPRs, where we store particular words of the operand A. This strategy saves several memory accesses for the load of the operand and reduces the stack usage.



The referred results are presented in:
[1] [SJA19]
[2] [SAJA20b]

Figure 12: Graph representation of the clock cycles, required for the execution of all SIKE primes ([1]does not provide implementation for SIKE p610) on STM32F407 @168MHz.

We separate our implementation into 2 different block types. On Figure 10 we present only the first block type that we used in our design. Due to the small length of the operands - 256 bits, the implementation of the entire squaring results in 2 blocks, both of SBD type. The sub-squaring block type is further split into front and back part where we calculate the double of the operand words with different indexes, and the single multiplication value of the equivalent indexes. We perform the SBD implementation block at the beginning and at the end of each row. We name them as $n\mathbf{F}$ and $n\mathbf{B}$, with n being the number of the row and **F** and **B** show the position of the squaring zone - at the front or at the back of the row, respectively. In Figure 11, the squaring implementation of all SIKE primes can be observed. Each of the rows starts and ends with the SBD blocks, where the product of the operand limbs with equal indexes are accumulated with the doubled product of the different indexes. We first calculate the multiplication of the different indexes products and at the end compute the value of the same index multiplication. We reuse the doubled words for the computation of the following columns and keep these doubled values inside the GPR set. Therefore, we need to re-load the value of the limbs in order to use them in their non-doubled form. However, due to our implementation strategy, using the FPR set as a level 1 cache, the reloading is performed in an efficient way, requiring again 1 CC. We observed that, opposite to the implementation in [SAJA20b], it is cheaper to load the non-doubled value inside the GPR set than to compute the doubled values, then store them into the stack and later obtain them back from the memory. The load and store operations require twice more cycles than the moving between different register sets. Therefore, our squaring implementation outperforms the previous designs due to the low access to the memory.

The second block type that we implement for our design is the sub-multiplication block, following the multiplication technique, described in [SAJA20b] and referred as a Refined-Operand Caching (R-OC). We implement the sub-multiplication between the SBD

Table 4: Comparison between the SIKE finite field arithmetic operations measured on STM32F407 @168MHz. The speedup obtained is marked in red color.

Implementation	Language	Timing [cc]/Speedup[%]							
		$\mathbb{F}_p$add	Speedup	$\mathbb{F}_p$sub	Speedup	$\mathbb{F}_p$mul	Speedup	$\mathbb{F}_p$sqr	Speedup
SIKEp434									
SIDH v3.3[1]	C	435	58.39	257	33.07	5,164	84.49	5,164	87.88
Seo et al.[2]	ASM	254	28.74	208	17.31	110	27.84	981	36.19
Seo et al.[3]	ASM	253	28.46	207	16.91	1,011	20.77	889	29.58
This work	ASM	181	-	172	-	801	-	626	-
SIKEp503									
SIDH v3.3[1]	C	492	59.55	327	41.90	6,778	85.48	6,778	88.70
Seo et al.[2]	ASM	275	27.64	223	14.80	1,333	26.18	1,139	32.75
Seo et al.[3]	ASM	274	27.37	227	16.30	1,254	21.53	1,060	27.74
This work	ASM	199	-	190	-	984	-	766	
SIKEp610									
SIDH v3.3[1]	C	667	64.47	455	49.89	1,0187	84.90	1,0187	88.19
Seo et al.[2]	ASM	331	28.40	272	16.18	1,898	18.97	1,573	23.52
This work	ASM	237	-	228	-	1538	-	1203	-
SIKEp751									
SIDH v3.3[1]	C	793	65.07	516	50.19	14285	85.05	14285	88.97
Seo et al.[2]	ASM	388	28.61	284	9.51	2,744	22.19	2,242	29.75
Seo et al.[3]	ASM	387	28.42	318	19.18	2,617	18.42	2,115	25.53
This work	ASM	277	-	257	-	2135	-	1575	-

The referred results are presented in:
[1] [SIK]
[2] [SJA19]
[3] [SAJA20b, SAJA20a]

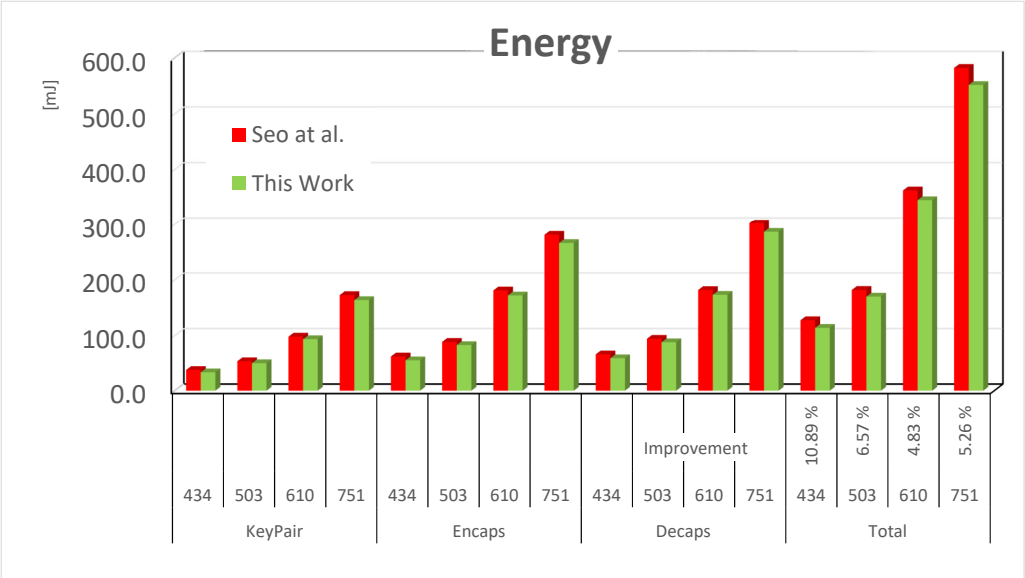
blocks. We mark them on Figure 11 as n **M**iddle-**F**ront (n**M-F**) and n **M**iddle-**B**ack (n**M-B**), where n is the row number. These rows are not computing words with equivalent indexes, therefore, the only difference from the multiplication R-OC technique is the doubling of one of the operands. However, since the doubled values in n**M-B** are reused from the n**M-F** block, the $\times 2$ operations there are avoided, optimizing the implementation. The proposed sequence of blocks, implementing different techniques, results in easy carry propagation, handled by the MAC instructions.

Our design outperforms the previous designs significantly by reconfiguring the sequence of operations. The performance of our optimal implementation is shown in the following section, where the speedup is due to the optimized block configuration and the use of FPR set as level 1 cache memory.

4 Performance Evaluations

In this section, we present the results that we obtained after applying the proposed optimization strategies. We have been focused on low-end devices, therefore, we have performed our experiments, targeting the processor Cortex-M4. We have used the boards STM32F407 Discovery Board, recommended by NIST as a low-end device, for benchmarking the clock cycles, the NUCLEO-F411RE and X-NUCLEO-LPM01A for measuring the power and energy consumption on Cortex-M4.

Our implementations shows significantly better results in comparison to the previous fastest implementation strategies. In Table 3, we have measured and reported the clock cycles required for the execution of the SIKE algorithm. For instance, our SIKEp434



The referred results are presented in [SAJA20b].

Figure 13: Graph representation of the energy consumption measured on NUCLEO-F4 and X-NUCLEO-LPM of all primes of SIKE.

implementations for NIST security level 1 takes about 850ms which is about 22.26% faster than the counterparts appeared in [SAJA20a].

We have obtained a speedup of 22.26%, 21.88% 19.66% and 19.54% for the primes SIKEp434, SIKEp503, SIKEp610 and SIKEp751, respectively, which is shown in Figure 12 a graphical representation of the speedup is presented. The considered comparison implementations are proposed by Seo et al. in [SJA19], marked in red color and [SAJA20b], marked in blue in the bar-chart graph. Both implementations are designed and implemented in assembly, therefore show the best performance for Cortex-M4 so far. The green bars present the performance or our opimized implementation, where the value of the speedup in percentage, compared to the previos best reported results [SAJA20a], is shown under the graph.

The pyramid-like computational structure of the SIKE operations ensures that the improvement of the underlying finite field operations will result in a speedup of the entire algorithm. In Table 4 the clock cycles required for the execution of each one of the finite field operations are reported before and after our proposed design. We improved the field addition by around 28% for all the SIKE primes the subtraction from 16% to 19%, the multiplication outperforms the previous implementation by 18.5% up to 21.5%, and the squaring shows up to 29.6% better results, in comparison to the best previous reported results by Seo at *al.* in [SAJA20b, SAJA20a].

The low energy and power consumption is main objective of the low-end processors, dedicated to the IoT world. They aim to be efficient not only in execution time but also to show small use of resources. We have measured the energy and power consumption using the NUCLEO-F4 board with frequency of 96 MHz. Table 5, reports the results we have obtained. It can be noticed that the energy consumption is decreased with 14 mJ, 12 mJ, 17 mJ and 31mJ for the SIKEp434, SIKEp503, SIKEp610 and SIKEp751, respectively. The results correspond to 11%, 6.6%, 4.7% and 5.3% of improvement of the energy consumption for the four prime numbers. Figure 13 illustrates the comparison in a bar-chart format, where the improvement of the energy consumption can be seen in the bottom right corner. The improvement of the energy consumption imposes an increase of

Table 5: Table representation of the energy and power consumption measured on NUCLEO-F4 and X-NUCLEO-LPM of all primes of SIKE.

Implementation	Language	Speed	Energy [mJ]				Power[mW]			
		[MHz]	KeyGen	Encaps	Decaps	Total	KeyGen	Encaps	Decaps	Total
SIKEp434										
SIDH v3.3[1]	C	96	485.00	798.32	850.72	1,649.04	66.91	67.28	67.20	134.48
Seo et al.[2]	ASM		37.26	61.60	65.54	127.14	73.74	74.70	74.34	149.04
This work	ASM		33.14	54.82	58.48	113.30	74.97	75.90	75.70	151.60
SIKEp503										
SIDH v3.3[1]	C	96	724.96	1,198.00	1,273.00	2,471.00	66.56	66.72	66.70	133.42
Seo et al.[2]	ASM		53.03	87.89	93.55	181.44	75.48	76.16	76.10	152.26
This work	ASM		49.58	82.18	87.34	169.52	79.79	80.62	80.40	161.02
SIKEp610										
SIDH v3.3[1]	C	96	1,358.00	2,516.00	2,528.00	5,044.00	66.63	66.06	66.98	134.04
Seo et al.[2]	ASM		97.36	180.50	181.30	361.80	78.17	78.75	78.74	157.49
This work	ASM		92.89	171.63	172.71	344.34	82.54	83.11	83.06	166.17
SIKEp751										
SIDH v3.3[1]	C	96	2,435.00	3,992.00	4,273.00	8,265.00	65.75	66.46	66.25	132.21
Seo et al.[2]	ASM		172.07	280.53	301.58	582.11	80.66	81.16	81.21	162.37
This work	ASM		163.22	265.73	285.79	551.52	84.74	85.24	85.31	170.55

The referred results are presented in:
[1] [SIK]
[2] [SAJA20b]

the power use. The proposed implementation increases the power consumption with few milliwatts, in particular 2 mW, 9 mW, 17 mW and 8 mW, respectively for the primes SIKEp434, SIKEp503, SIKEp610 and SIKEp751. This increase is equivalent to 1.3%, 5.9%, 5.7% and 4.9% of power increase, which is insignificant taking into consideration the performance boost as well as energy reduction. Also, it should be noted that for battery-powered devices energy consumption is the most critical parameter.

5 Conclusion

In this work we presented a highly optimized implementation of the SIKE underlying finite field arithmetic operations. Our target platform is the low-end processor Cortex-M4, recommended by NIST for benchmarking the PQC algorithms.

We propose several novel ideas for the modular addition, multiplication, squaring and reduction techniques. Using the computational block alternating strategy for the modular addition, the number of carry/borrow catcher operations is reduced. Moreover, we propose to modify the value of the prime number, where due to its special form, this strategy results in the elimination of several subtraction instructions. We improve the implementation of the multiplication, squaring and reduction by using the FPR set as a level 1 cache memory, which we used to avoid the memory access instructions. We propose two different multi-precision multiplication strategies, depending on the length of the prime. We use the FPR set to either store the values of the operands - therefore, to avaid all memory accesses for loading their lims, or to store the partial results produced by each row - thus, to reduce the stack usage. Moreover, we proposed novel reduction and squaring techniques, which use new order of instructions and hence show significant time and energy optimizations. Our implementation for the addition, subtraction, multiplication and squaring outperforms the other implementation for p434 with 28.46%, 16.91%, 20.77% and 29.58%, respectively.

We have proposed an implementation design for the underlying finite field arithmetic operations, ensuring the fast protocol performance. Our improvements of the modular

addition, subtraction, multiplication and squaring result in significantly better overall protocol performance compared to the previous best implementation, where for p434 the performance improvement is 22.3%. The optimizations also show better energy consumption, where for p434 it results to be 10% more efficient.

We hope to push SIKE further in the PQC NIST competition after the implemented optimizations, since it is the candidate with the smallest key sizes, therefore, ensures insignificant communication latency. We are going to continue our effort to constantly improve the timing of the post-quantum algorithm, where we are willing to perform side-channel analysis of our implementations as a future project.

References

[Ber09] Daniel J Bernstein. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, pages 1–14. Springer, 2009.

[CLN16] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. In *Annual International Cryptology Conference*, pages 572–601. Springer, 2016.

[DF] Luca De Feo. Mathematics of isogeny based cryptography. corr, abs/1711.04062, 2017. https://arxiv.org/pdf/1711.04062.pdf. Last accessed on January 2, 2021.

[DSS16] Fabrizio De Santis and Georg Sigl. Towards side-channel protected x25519 on arm cortex-m4 processors. *Proceedings of Software performance enhancement for encryption and decryption, and benchmarking, Utrecht, The Netherlands*, pages 19–21, 2016.

[EAMK20a] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Efficient and fast hardware architectures for sike round 2 on fpga. Technical report, Cryptology ePrint Archive 2020/611, 2020.

[EAMK20b] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Highly optimized montgomery multiplier for sike primes on fpga. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 64–71. IEEE, 2020.

[FA17] Hayato Fujii and Diego F Aranha. Curve25519 for the cortex-m4 and beyond. In *International Conference on Cryptology and Information Security in Latin America*, pages 109–127. Springer, 2017.

[HL19] Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based pake protocol tailored for the iiot. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–48, 2019.

[HW11] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 459–474. Springer, 2011.

[JAC+17] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular Isogeny Key Encapsulation. Submission to the NIST Post-Quantum Standardization Project, 2017.

[KAEK+20] Brian Koziel, A-Bon Ackie, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Sike'd up: Fast hardware architectures for supersingular isogeny key encapsulation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[KJA+16] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. 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, 2016.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[KPHS18] Philipp Koppermann, Eduard Pop, Johann Heyszl, and Georg Sigl. 18 seconds to key exchange: Limitations of supersingular isogeny diffie-hellman on embedded devices. *IACR Cryptol. ePrint Arch.*, 2018:932, 2018.

[LG14] Zhe Liu and Johann Großschädl. New speed records for montgomery modular multiplication on 8-bit avr microcontrollers. In *International Conference on Cryptology in Africa*, pages 215–234. Springer, 2014.

[NAK20] Mojtaba Bisheh Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Efficient hardware implementations for elliptic curve cryptography over curve448. In *International Conference on Cryptology in India*, pages 228–247. Springer, 2020.

[NEKAMK20] Mojtaba Bisheh Niasar, Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast, small, and area-time efficient architectures for key-exchange on curve25519. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 72–79. IEEE, 2020.

[oSN] The National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization, 2017â2018. https://csrc.nist.gov/projects/post-quantum-cryptography/ post-quantum-cryptography-standardization. Last accessed on January 10, 2021.

[PLW+20] Jun-Hoe Phoon, Wai-Kong Lee, Denis Chee-Keong Wong, Wun-She Yap, and Bok-Min Goi. Area–time-efficient code-based postquantum key encapsulation mechanism on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2672–2684, 2020.

[PQC] PQCryptov3.2. Sidh library. https://github.com/microsoft/PQCrypto-SIDH/releases/tag/v3.2.

[SAJA20a] H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh. Supersingular isogeny key encapsulation (sike)round 2 on arm cortex-m4. *IEEE Transactions on Computers*, (to appear):1–1, 2020.

[SAJA20b] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (sike) round 2 on arm cortex-m4. *IACR Cryptol. ePrint Arch.*, 2020:410, 2020.

[Seo20] Hwajeong Seo. Memory efficient implementation of modular multiplication for 32-bit arm cortex-m4. *Applied Sciences*, 10(4):1539, 2020.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[SIK] SIKE. Sike website. https://sike.org/. Last accessed on January 5, 2021.

[SJA19] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. Sike round 2 speed record on arm cortex-m4. In *International Conference on Cryptology and Network Security*, pages 39–60. Springer, 2019.

[SK12] Hwajeong Seo and Howon Kim. Multi-precision multiplication for public-key cryptography on embedded microprocessors. In *International Workshop on Information Security Applications*, pages 55–67. Springer, 2012.

[SK15] Hwajeong Seo and Howon Kim. Consecutive operand-caching method for multiprecision multiplication, revisited. *Journal of information and communication convergence engineering*, 13(1):27–35, 2015.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. Sidh on arm: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–20, 2018.

[SSJA20] Hwajeong Seo, Pakize Sanal, Amir Jalali, and Reza Azarderakhsh. Optimized implementation of sike round 2 on 64-bit arm cortex-a processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.

[Tat66] John Tate. Endomorphisms of abelian varieties over finite fields. *Inventiones mathematicae*, 2(2):134–144, 1966.

[Vél71] Jacques Vélu. Isogénies Entre Courbes Elliptiques. *Comptes Rendus de l'Académie des Sciences Paris Séries A-B*, 273:A238–A241, 1971.