Compressed SIKE Round 3 on ARM Cortex-M4

Mila Anastasova[1], Mojtaba Bisheh-Niasar[1], Reza Azarderakhsh[1,2] and Mehran Mozaffari Kermani[3]

[1] Computer and Electrical Engineering and Computer Science Department and I-SENSE at Florida Atlantic University, Boca Raton, FL, USA
`(manastasova2017, mbishehniasa2019, razarderakhsh)@fau.edu`
[2] PQSecure Technologies, LLC, Boca Raton, FL, USA
[3] Computer Engineering and Science Department at University of South Florida, Tampa, FL, USA,
`mehran2@usf.edu`

Abstract. In 2016, the National Institute of Standards and Technology (NIST) initiated a standardization process among the post-quantum secure algorithms. Forming part of the alternate group of candidates after Round 2 of the process is the Supersingular Isogeny Key Encapsulation (SIKE) mechanism which attracts with the smallest key sizes offering post-quantum security in scenarios of limited bandwidth and memory resources. Even further reduction of the exchanged information is offered by the compression mechanism, proposed by *Azarderakhsh et al.*, which, however, introduces a significant time overhead and increases the memory requirements of the protocol, making it challenging to integrate it into an embedded system. In this paper, we propose the first compressed SIKE implementation for a resource-constrained device, where we targeted the NIST recommended platform STM32F407VG featuring ARM Cortex-M4 processor. We integrate the isogeny-based implementation strategies described previously in the literature into the compressed version of SIKE. Additionally, we propose a new assembly design for the finite field operations particular for the compressed SIKE, and observe a speedup of up to 16% and up to 25% compared to the last best-reported assembly implementations for p434, p503, and p610.

Key Words: Compressed Supersingular Isogeny Key Encapsulation (SIKE), Post-Quantum Cryptography (PQC), ARM Cortex-M4

1 Introduction

Public key cryptography is essential for the confidentiality and integrity of data transmitted through an insecure channel. The classical schemes used nowadays such as RSA and the ECC family, however, are going to be broken when a large-scale quantum computer is developed. Driven by the technology progress the National Institute of Standards and Technology (NIST) [1] initialized a standardization process aiming at optimizing and evaluating the post-quantum candidates. In 2020, Round 3 of the competition has started, where the post-quantum secure Key Encapsulation Mechanisms are divided into two subgroups – finalists and alternate candidates. Forming part of the alternate candidates is the only isogeny-based cryptosystem – Supersingular Isogeny Key Encapsulation (SIKE) mechanism, which, based on the Supersingular Isogeny Diffie-Hellman (SIDH) algorithm [2], attracts with the smallest public key and ciphertext sizes (i.e., 330 and 346 bytes for the NIST security level 1 implementation). Thus, it ensures negligible communication latency of the algorithm and comes into use in a bandwidth-limited environment.

In 2016, *Azarderakhsh et al.* proposed even further reduction of the transmitted information in [3] where the team reduced the size of the data by a factor of 2 by applying a novel idea for compression mechanism. The significant timing overhead introduced when using the compression incited optimizations in the computational cost of the scheme presented in [4], [5], and [6]. The compression mechanism, however, still introduces a non-negligible timing overhead and requires more available memory, making the protocol execution difficult on low-end devices which offer limited resources.

The compressed SIKE is the algorithm with the closest communication latency to today's cryptosystems. The extremely compact key sizes offered by the Elliptic Curve Cryptography (ECC) family lead to multiple optimizations of the algorithm targeting software and hardware [7], [8], [9], [10] reporting indisputable time and energy efficiency. Nevertheless, despite the inexpensive computational cost and the extremely low key sizes, this classical crypto scheme will not ensure secure data transmission in the scenario of large quantum computers which leads to the NIST post-quantum standardization effort, where the complexity of some algorithms, such as SIKE, introduces higher computational cost. Thus, several researchers have centered their work on improving the efficiency of the PQ scheme on software [11], [12], [13], [14], [15], targeting ARMv7-M, ARMv7-A and ARMv8 ARM-based architectures, and hardware [16], [17], [18], [19] targeting the Xilinx Virtex 7 FPGA family hardware platform.

Contribution. In this work, we propose the first implementation of compressed SIKE, targeting the resource-restricted processor ARM Cortex-M4. We integrate the finite field strategies proposed in [12] and [11] into the compressed SIKE implementation, we hand-code additional subroutines and report the achieved results. Our contributions are itemized as follows:

- We propose optimal usage of the memory, increasing the size of the stack to 128KB and creating a memory region in the Core Coupled Memory (CCM RAM) increasing the RAM from 112KB as proposed in PQM4 [20] to 192KB. For the correct execution of compressed SIKEp610, we split large structures residing in the memory into parts and allocate them in the stack and the CCM RAM ensuring that the program does not overwrite illegal memory regions exceeding the size of the stack.
- We implement additional subroutines particular for compressed SIKE in an efficient assembly language - multi-precision subtraction with correction $2p$ and $4p$ and multi-precision multiplication of 256×256- and 320×320-bit length, where we apply the strategies described in the isogeny-based works [12] and [11].

Public Parameters: $p = 2^{e_A}3^{e_B} - 1$, $E_0/\mathbb{F}_{p^2}$, $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$		
Alice		**Bob**
Input: -		
Output: s, sk_A, pk_A		
1. $sk_A \in_R \mathbb{Z}/2^{e_A}\mathbb{Z}$		
2. $\phi_A : E_0 \rightarrow E_A$ with		
$ker(\phi_A) = \langle P_A + [sk_A]Q_A \rangle$		**Input:** pk_A
3. $pk_A = (E_A, \phi_A(P_B), \phi_A(Q_B))$		**Output:** c, ss
4. $s \in_R \{0,1\}^t$		**Decompress public key**
Compress public key		1. $m \in_R \{0,1\}^t$
Input: s, sk_B, pk_B, c		2. $r = H(m\|\|pk_A) mod 3^{e_B}$
Output: ss	$\xrightarrow{pk_A}$	3. $\phi_B : E_0 \rightarrow E_B$ with
Decompress ciphertext		$ker(\phi_B) = \langle P_B + [r]Q_B \rangle$
1. $\phi'_A : E_B \rightarrow E_{BA}$ with	$\xleftarrow{c}$	4. $pk_B = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$
$ker(\phi'_A) = \langle \phi_B(P_A) + [sk_A]\phi_B(Q_A) \rangle$		5. $\phi'_B : E_A \rightarrow E_{AB}$ with
2. $m' = c_1 \oplus K(j(E_{BA}))$		$ker(\phi'_B) = \langle \phi_A(P_B) + [r]\phi_A(Q_B) \rangle$
3. $r' = H(m'\|\|pk_A) mod 3^{e_B}$		6. $c = (c_0, c_1) = (pk_B, K(j(E_{AB})) \oplus m)$
4. $\phi''_A :\ E_0 \rightarrow E_{B'}$ with		7. $ss = (J(m\|\|c))$
$ker(\phi''_A) = \langle P_B + [r']Q_B \rangle$		**Compress ciphertext**
5. $pk'_B = \{E_{B'}, \phi''_A(P_A), \phi''_A(Q_A)\}$		
6. IF $pk'_B = pk_B$		
$ss = (J(m'\|\|c))$		
ELSE $ss = (J(s\|\|c))$		

Fig. 1. SIKE algorithm [23]. H, K and J denote hash functions.

– We integrate the previous best-reported arithmetic operations optimizations proposed in [12] and [11] and report a speedup of up to 25% and up to 16%, respectively, after integrating the new subroutine implementation for p434, p503, and p610 compressed SIKE primes.

2 Preliminaries

This section presents a detailed description of the (un)compressed SIKE protocol and a description of the target platform. For a more comprehensive description of the algorithms refer to [21] and [22].

2.1 SIKE

The Supersingular Isogeny Key Encapsulation mechanism forms part of the alternate candidates after NIST Round 2, thus it is still going through significant improvements for the high- and low-end target devices. In this work, we present the first design of the compressed SIKE protocol targeting ARM Cortex-M4 processor.

Detailed graphical representation of the steps performed by the communication parties during the execution of the SIKE protocol is shown in Fig. 1 where they are described as follows:

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)	Public Key Size (B)	Cipher Text Size (B)
		Standard		Compressed	
`SIKEp434`	1	330	346	197	236
`SIKEp503`	2	378	402	225	280
`SIKEp610`	3	462	486	274	336
`SIKEp751`	5	564	596	335	410

Table 1. Round 2 SIKE public parameters [23].

- Public Parameters: Alice and Bob start from supersingular elliptic curve $E_0/\mathbb{F}_{p^2}$ with prime number $p = 2^{e_A}3^{e_B} - 1$ along with basis points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ that generate $E_0[2^{e_A}]$ and $E_0[3^{e_B}]$, respectively.
- Key Generation: Alice computes a random integer $sk_A \in_R \mathbb{Z}/2^{e_A}\mathbb{Z}$ and uses it to compute a secret isogeny ϕ_A, where she uses the image curve E_A and the image points $\phi_A(P_B), \phi_A(Q_B)$ to form her public key pk_A.
- Encapsulation: Bob computes his secret key r based on Alice's public key and a random message m, which he uses to compute the kernel of his secret isogeny and to form his public key as $pk_B = \{E_B, \phi_B(P_A), \phi_B(Q_A)\}$. He computes a second isogeny, which leads him to the final image curve E_{AB}. He generates a ciphertext appending the message m masked by $j(E_{AB})$ of the curve to his public key.
- Decapsulation: Alice uses her secret key and Bob's public key projection points to reach curve E_{BA}, which since it is isomorphic to E_{AB} features the same $j-$invariant. She uses the unmasked value m to obtain Bob's secret key and uses it to re-compute his public key, where by comparing the result with c_0, she computes the shared secret or uses random value, preventing further communication.

2.2 Compression Mechanism

The insignificant bandwidth required for the execution of SIKE motivated several research teams to work and further optimize the size of the exchanged data. As shown in Table 1, there is a considerable difference between the key sizes of the standard and the compressed version of SIKE, which features the smallest key sizes among the post-quantum candidates even before the compression mechanism is applied.

The key compression mechanism has been proposed in 2016 and, due to the applications of compressed SIKE into the IoT world where the resources are strictly limited, it has gone through several modifications aiming at improving the main drawback of the protocol – the performance. In Fig. 2, the compression and decompression of the public information are shown, where the actual content of the communicated information has been modified with the improvements introduced by several research groups. Nevertheless, the compression mechanism requires the execution of three main steps: basis generation, pairing and discrete logarithms computations, where these phases introduce a non-negligible overhead to the execution of the algorithm.

The compression mechanism, proposed by *Azarderakhsh et al.* [3], reduces the size of the Supersingular Isogeny Diffie-Hellman, base of SIKE, public information by a factor of 2 – from $8log_2p$ to $4log_2p$. The improvement is achieved by replacing the previous public key tuple $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ by new representation $\{j(E_A) \in \mathbb{F}_{p^2}, a_1, a_2, b_1, b_2 \in \mathbb{Z}_{3^{e_3}}\}$ with the $j-$invariant of the curve and four integers such that $\phi_A(P_B) = a_1R_1 + a_2R_2$ and $\phi_A(Q_B) = b_1R_1 + b_2R_2$ for canonical basis $\{R_1, R_2\}$ for $E_A[3^{e_3}]$. However, the additional computation of new points of order 3^{e_3}, the pairing algorithm revealing that the points actually form basis for $E_A[3^{e_3}]$ and discrete logarithms allowing the representation of the two image points with 4 coordinates in $\mathbb{F}_{p^2}$ using only 4 values in $\mathbb{Z}_{3^{e_3}}$, introduces and overhead of more than $10\times$ the execution time of the uncompressed algorithm.

Costello et al. have proposed several computational optimizations of the compression algorithm in [4], where they additionally achieve further key size reduction to $\frac{7}{2}log_2p$ by sending $\{j(E_A) \in \mathbb{F}_{p^2}, \alpha, \beta, \gamma \in \mathbb{Z}_{3^{e_3}}\}$ when normalizing three of the elements with a deterministically chosen invertible element. Their computation improvements are based on efficient torsion basis generation, fast Tate pairing [24], [25] (replacing the Weil pairing), SIDH specific inversion-free pairing formulas, and highly optimized formulas for solving discrete logarithms. The improvements made by the authors decrease the performance overhead of the compression mechanism which leads to the integration of the proposed key size optimizations into the NIST standardization process SIDH protocol proposal.

In [26], *Zanon et al.* have proposed the use of entangled basis generation, which has further application in the isogeny-based hash functions and is proved to be more than $15\times$ more efficient than the usual basis generation. They achieve further performance improvement of the decompression by using shared elligator technique and reverse basis decomposition. The authors continue with the optimizations by applying an *optimal strategy* for the discrete logarithms and by exploiting particular characteristics of the entangled basis that speed up the Tate pairing computation.

Later, *Naehrig et al.* [5] have reduced the compression overhead by integrating dual isogenies to pull back the deterministically generated basis points to the original elliptic curve E. The use of dual isogeny $\hat{\phi}$ is applied in the basis generation and the pairing step, where the Tate pairings evaluate $\tau_{3^{e_3}}(P_B, \hat{\phi_A}(R_1))$, $\tau_{3^{e_3}}(P_B, \hat{\phi}_A(R_2))$, $\tau_{3^{e_3}}(Q_B, \hat{\phi}_A(R_1))$ and $\tau_{3^{e_3}}(Q_B, \hat{\phi}_A(R_2))$ instead of $\tau_{3^{e_3}}(\phi_A(P_B), R_1)$, $\tau_{3^{e_3}}(\phi_A(P_B), R_2)$, $\tau_{3^{e_3}}(\phi_A(Q_B), R_1)$ and $\tau_{3^{e_3}}(\phi_A(Q_B), R_2)$, leading to significant speedup since the computations are pulled back to the fixed starting curve and no changes throughout different executions of the protocol occur.

This work has been continued by *Pereira et al.* in [6] where they improve further the shared elligator technique, which allows to deterministically find a point on an elliptic curve, by integrating another two bits of information into the public key, showing the correct ternary basis generators. The implementation is

Public Key (De)Compression

1. Compress Public Key

$\langle R_1, R_2\rangle = E_A\,[3^{e_B}]$
$c_0, c_1 : \phi_A(P_B) =$
$[c_0]\hat{\phi}_A(R_1) + [c_1]\hat{\phi}_A(R_2)$
$d_0, d_1 : \phi_A(Q_B) =$
$[d_0]\hat{\phi}_A(R_1) + [d_1]\hat{\phi}_A(R_2)$
IF $d_1 mod 3^{e_B} = 0$
$\alpha = -d_0^{-1}d_1,\ \beta = c_1 d_0^{-1},\ \gamma = -c_0 d_0^{-1}$
ELSE
$\alpha = -d_0 d_1^{-1},\ \beta = -c_1 d_1^{-1}, \gamma = c_0 d_1^{-1}$
$pk_A = pk\_comp_A = \{\alpha, \beta, \gamma, A\}$

$\xrightarrow{pk_A}$

2. Decompress Public Key

$\langle R_1, R_2\rangle = E_A\,[3^{e_B}]$
IF $d_1 mod 3^{e_B} = 0$
$ker(\phi'_B) =$
$\langle R_1 + [((\alpha + [r]\gamma)(1 + [r]\beta))^{-1}]R_2\rangle$
ELSE
$ker(\phi'_B) =$
$\langle R_1 + [((\alpha + [r]\beta)(1 + [r]\gamma))^{-1}]R_2\rangle$

Ciphertext (De)Compression

3. Compress Ciphertext

$\langle S_1, S_2\rangle \in E_B\,[2^{e_A}]$
$a_0, b_0 : \phi_B(P_A) =$
$[a_0]\hat{\phi}_B(S_1) + [b_0]\hat{\phi}_B(S_2)$
$a_1, b_1 : \phi_B(Q_A) =$
$[a_1]\hat{\phi}_B(S_1) + [b_1]\hat{\phi}_B(S_2)$
$c_0 = (A, (a_0, b_0, a_1, b_1))$

$\xleftarrow{c}$

4. Decompress Ciphertext

$\langle S_1, S_2\rangle \in E_B\,[2^{e_A}]$
$\phi'_A : E_B \to E_{BA}$ with
$ker(\phi'_A) =$
$\langle S_1 + [(sk_A a_1 + a_0)^{-1}(sk_A b_1 + b_0)]S_2\rangle$
or
$ker(\phi'_A) =$
$\langle S_1 + [(sk_A b_1 + b_0)^{-1}(sk_A a_1 + a_0)]S_2\rangle$

Fig. 2. Public Key/Ciphertext (De)Compression. For more details refer to [22].

performed by sharing two counter variables, generated during the compression phase, which allow the decompression step to be straightforward executed finding the correct basis points of the curve by accessing specific entry of precomputed tables. The authors also propose the use of $x-$only point addition formula which results in a much more efficient decompression step. The paper also proposed an increase of the ciphertext size (eliminating the previous reduction proposed in [4]), showing a trade-off between transmitted data size and computational overhead, which makes the comparison between the transmitted key and the re-computed one easier and results in a speedup of the decapsulation phase.

Finally, *Hutchinson et al.* [27] propose techniques for reducing the pairing and discrete logarithm tables by a factor of 4 by signed digit representation of exponents and torus-based representation of cyclotomic subgroup elements. These techniques introduce a slight overhead, however, are crucial for the integration of compressed SIKE into low-end devices with limited resources.

A detailed description of the compression and decompression steps taken by Alice and Bob are presented in Fig. 2 which are described as follows:

- Public key compression: Alice computes the unique dual isogeny $\hat{\phi}_A$ of her secret isogeny map which she uses to pull back the canonical basis points R_1, R_2, generating $E_A\,[3^{e_B}]$, to the elliptic curve with Mongtomery coeffiecient equal to 0 such that $\hat{\phi}_A(R_1), \hat{\phi}_A(R_2) \in E_{A'=0}$. She then computes 4 discrete logarithms, using optimal Pohlig-Hellman [28] strategy, c_0, c_1, d_0, d_1 and finally obtains the value of $\alpha, \beta, \gamma \in (\mathbb{Z}_{3^{e_3}})^3$, that form part of the compressed public key replacing $\phi_A(P_B)$ and $\phi_A(Q_B)$.

– Public key decompression: Bob recovers the value of the canonical basis points R_1, R_2 and uses them along with the triplet from Alice's public key to compute the kernel of his second isogeny.
– Ciphertext compression: Bob computes canonical basis points S_1, S_2 to compress his public key, which forms part of the ciphertext. For efficiency reasons [6] the triplet is not computed but rather a linear combination of the four values of a_0, a_1, b_0, b_1 are sent to Alice, introducing a tradeoff between ciphertext length and decompression (thus decapsulation) execution time.
– Ciphertext decompression: Alice computes the canonical basis S_1, S_2 and uses the four values sent by Bob to compute her second isogeny and to reach the elliptic curve $\phi'_A : E_B \to E_{BA}$, featuring the same $j-$invariant as E_{AB} which helps her to recover the value of the masked message m.

2.3 ARMv7-M Architecture

The fast development of the Internet of Things world results in extremely high demand for low-end devices, which can be integrated into any real-time embedded system. Thus, NIST has announced the Reduced Instruction Set Computer (RISC) ARM Cortex-M4-based microcontroller STM32F407VG as the target microcontroller for the implementation and benchmarking of the post-quantum secure schemes on resource-constraint devices. The standardization effort has incited the implementation of the PQM4 library [20], which integrates test and benchmark framework used in this work to perform the measurements.

The STM32F407VG microcontroller features 1MB of flash memory and 192KB of RAM. However, the memory map of the device Fig. 3 shows the division of the RAM into 3 different memory blocks – 2 consecutive SRAM blocks of 112KB and 16KB, and 1 Cored Coupled Memory (CCM RAM) block of 64KB in a separate memory region. In PQM4, for efficiency reasons, only the first SRAM memory region is used for the execution of SIKE, thus the resources of the device are brought down to only 112KB, which results to be insufficient for the execution of the compressed SIKE protocol.

The 3-stage pipeline of the ARMv7-M architecture allows the fast execution of the instruction set, where most of the instructions require a single clock cycle. However, the load/store design requires another extra cycle for the completion of memory access instructions. In some particular cases, the instructions can be scheduled, thus the additional cycle is absorbed by the following instruction, however, depending on the nature of the implemented crypto scheme, this scheduling technique may be or may not be feasible to apply.

The platform, based on ARMv7-M architecture, features 16 32-bit General Purpose Registers (GPRs) and another 32 32-bit Floating-Point Registers (FPRs). Two of the GPRs are reserved for the Stack Pointer and the Program Counter, therefore, cannot be modified by the programmer, whereas the use of all FPRs is allowed, providing another 1024 bits of register space. Furthermore, the access cost for the information stored in the FPRs consists of a single clock

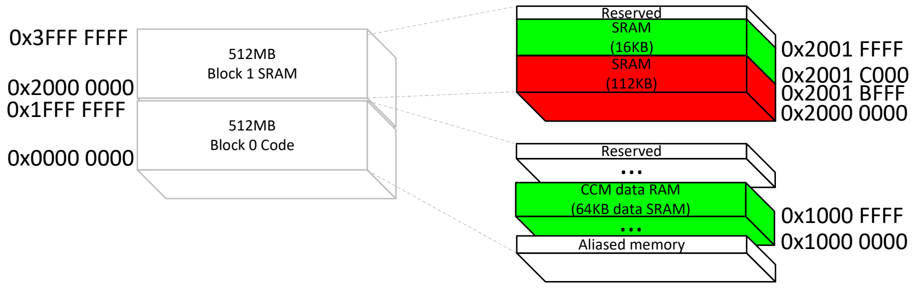

Fig. 3. Memory map of STM32F407VG device. The memory region used in PQM4 library [20] for the execution of SIKE is marked in red.

cycle. The `VMOV` instruction ensures instant data transfer between the two register sets, where it can be used to replace memory accesses, reducing the cost of `LDR` and `STR` instructions.

The use of Multiply ACcumulate (MAC) instructions has been the focus of several researchers in the last years due to the execution of up to 3 operations in a single clock cycle, thus we have integrated the use of `UMULL` and `UMAAL` into our work.

The procedure call standard for ARM architecture [29] states that the core registers `R4-R11` and floating-point registers `S16-S31` are not callee-saved, thus their value should be preserved among function calls by (v)pushing them in the stack at the beginning of a subroutine and (v)popping them at the end of a function call.

3 Compressed SIKE on STM32F407VG

The resource-constrained target platform and the PQM4 library settings did not allow the execution of the compressed SIKE for security levels 3 and 5. We applied some changes to the architecture of the linker file to make possible the execution of compressed SIKEp610 on the STM32F407VG platform. Furthermore, we implemented finite-field and multi-precision arithmetic particular for compressed SIKE in assembly language to speed up further the execution time of the protocol.

3.1 Configuration Modifications

The PQM4 library, base of our benchmarking, provides an implementation of the uncompressed SIKE, however, does not integrate the compressed SIKE due to the resource limitations of the target platform. The library configurations for SIKE are designed to fit the needs of the crypto scheme eliminating slow SRAM regions overhead. Specifically, for the execution of SIKE, the authors of the library integrate the use of only the first 112KB of SRAM, where the following 16KB are not in use due to efficiency reasons.

The execution of the compressed SIKE, however, requires large data structures, needed for the dual isogeny computation, which sizes increase with the number of isogeny maps needed thus with the security level of the algorithm. For the primes p610 and p751 the 112KB of SRAM were not enough to store the allocated local tables and resulted in stack overflow, causing the program to break due to overwriting illegal memory regions.

To allow the correct execution of compressed SIKEp610, we increased the size of the stack including the extra 16KB of SRAM by modifying the linker file. However, a total of 128KB of SRAM was still not enough for the compressed SIKEp610 execution. Therefore, we created a new memory region, which we placed inside the CCM RAM memory. We had to split the large dynamically allocated data structures into parts and place them separately into the SRAM (the stack) or the CCM RAM region. The CCM RAM memory is primarily designed for running code fast (with zero wait states) thus, placing some variables in this region may result in a speedup of the execution without any modification on the algorithm design. However, for precise measurement results, the PQM4 [20] benchmark framework suggests running the code @24MHz which eliminates all wait states and ensures an accurate number of clock cycles. Thus, the performance improvements discussed in this work are solely a result of the hand-coded assembly subroutines described in the following sections. The memory increase allowed us to execute the compressed SIKEp610 algorithm, where we had to perform some slight changes to the code, splitting the data structures and addressing them accordingly during the execution. The compressed SIKEp751 requires more memory than the entire 192KB of RAM, thus, it is not the focus of this work.

3.2 Field Subtraction and Multi-Precision Multiplication

In this work, we integrated the arithmetic operations underlying SIKE described in [12] and [11] into the implementation of compressed SIKE. The compressed crypto scheme, however, requires other functions, specific to the compression mechanism of SIKE. We implemented them in assembly language, integrating the strategies described in the literature, in particular, in [12] and [11], where, due to the performance optimizations and the invocation rate of the subroutines, we end up with a significant speedup.

Field Subtraction The importance of long integer modular addition and subtraction for cryptography is undeniable and its high invocation rates incite continuous improvement of these arithmetic operations [30], [12], [11]. The implementation of long integer addition/subtraction requires proper management of the carry/borrow propagation among the different limbs of the operands. The integration of correction step in the subtraction subroutine, bringing the integer back into the finite field specified by the security level implementation of the protocol, implies the execution of one extra long-integer addition, where in the scope of the compressed SIKE there are two different functions – subtract with

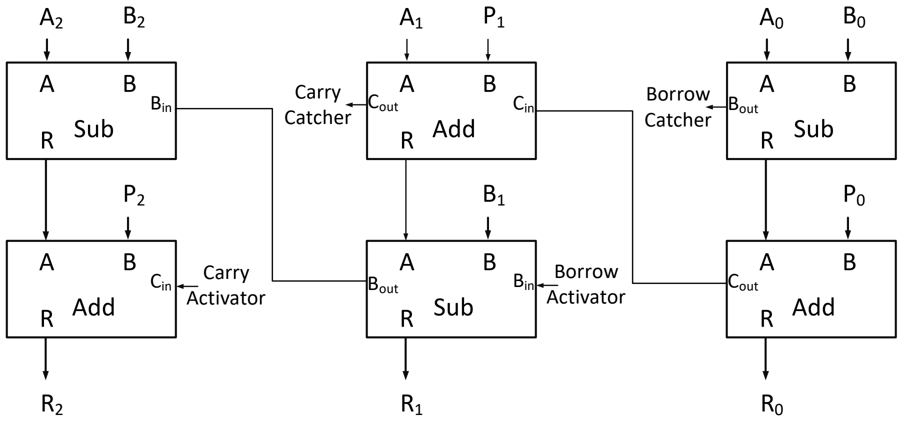

Fig. 4. Subtraction with correction P, where P represents $2p$ or $4p$.

correction $2p$ and $4p$, performing $a - b + 2p$ and $a - b + 4p$, respectively. We implement these functions in an efficient assembly code using the implementation strategies described in [11].

First, we apply the blocking strategy, where we split the long integer into several parts as shown in Fig. 4. We perform the subtraction and addition steps to each block managing the carry/borrow propagation with the help of carry/borrow catcher/activator, using the reduced instruction set `SBC` and `RSBC`. This strategy allows to reduce the number of registers used for the carry/borrow management and to increase the block size from 4 words per block to 5. Thus, the memory access latency is reduced along with the number of multiple load/store instructions which ensures scheduled code and minimal overhead.

Second, we integrate the block operation alternation, where we flip the applied add/sub instructions as shown in Fig. 4 among consecutive blocks allowing to manage the carry/borrow propagation using simply the `ADCS` / `SBCS` instructions. Therefore, we reduce the number of carry/borrow catchers/activators and use fewer instructions per modular subtraction.

Multi-Precision Multiplication Due to the extremely high invocation rate of the multi-precision multiplication in the scope of SIKE and compressed SIKE, its fast implementation becomes crucial for the crypto scheme's performance. Several research groups have been working on optimizing the multi-precision multiplication function by applying the Karatsuba method [30], [31], focusing on the powerful MAC instruction set [32], [33] and reducing the memory accesses by reusing the loaded values in the register set as long as possible [34], [12].

The implementation of compressed SIKE integrates multiplication subroutines of smaller sizes than the length of the primes, due to the new values contained into the public key and the ciphertext messages. In particular, the compressed SIKE uses multi-precision multiplication of two operands consisting

Table 2. Compressed SIKE arithmetic performance cost @24MHz.

Implementation	Timing [cc×10^6]								
	sub_2p	sub_4p	mul	sub_2p	sub_4p	mul	sub_2p	sub_4p	mul
	`SIKEp434 compressed`			`SIKEp503 compressed`			`SIKEp610 compressed`		
SIDH v3.3[1]	577	579	4,940	656	656	4,940	813	813	8,034
This work	135	135	202	147	147	202	180	180	296

[1] [35]

of 8 32-bit words for p434 and p503 and 10 32-bit words for p610. The multiplication results are not reduced, therefore, the outputs are two times longer than the operand sizes. In Fig. 5, we describe the implementations of both lengths of multi-precision multiplications using the visual rhombus representation where every diagonal line denotes a limb from the operand. The white dots show a 32×32-bit multiplication of the words from the operands corresponding to the crossing diagonals. The stroked white fields represent the consecutive accumulative multiplications and are referred to as rows in the literature. Finally, the red numbers denote the floating-point registers storing the partial results, where they are used in a consecutive order increasing from right to left. To reduce the execution timing of the multi-precision multiplication and thus to increase the efficiency of the entire protocol, we implemented the functions using assembly code as proposed in [12] and [11]. We used the Multiply ACcumulate instructions `UMULL` and `UMAAL`, implementing the Refined-Operand Caching design proposed in [12] with a maximum row size equal to 4. We implemented $256 \times 256-$ and $320 \times 320-$bit multiplications, where we use the FP register set as a cache memory, storing the partial result values resulting from the different row calculations as shown in Fig. 5, using the low-cost `VMOV` instruction.

Analysis of the Performance The implementation of all the finite field operations is performed in constant time, including the work in [12] and [11], which has been adapted to the compressed SIKE algorithm in the scope of this work. We have performed our measurements, as described in Section 4, running the algorithm @24MHz and @168MHz, for achieving a precise number of clock cycles and for reporting the minimal timing that the protocol can be executed, respectively. The underlying arithmetic operations are the reason for the performance speedup of the entire protocol due to the pyramid-like structure of the isogeny-based post-quantum scheme. In Table 2, we report the obtained performance after we have implemented the subtraction and the multiplication functions in assembly language. We compare our performance with the previous best (and only) implementation of the given subroutines which we have obtained from [35] implemented in portable C language. The comparison table shows that the target-specific implementation of the subtraction and multiplication is improved by 76.60% and 95.91%, respectively, for p434, 77,59% and 95.91% for p503, and 77.86% and 96.32% for p610. The global performance of compressed SIKE is

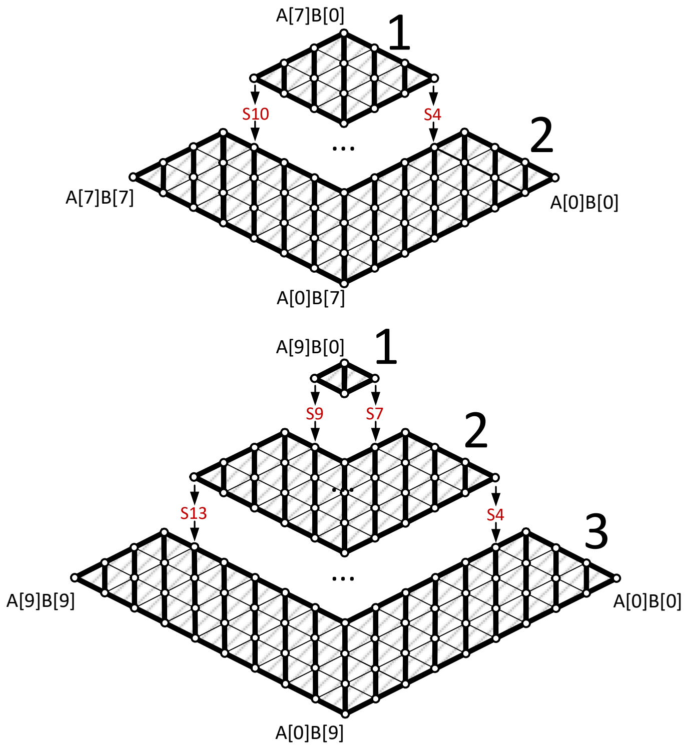

Fig. 5. $256 \times 256-$ and $320 \times 320-$ bit multiplication, using the FPRs as cache level 1 storing partial result values.

improved significantly by the implementation optimizations of the functions but also due to the high invocation ratio of these arithmetic operations.

4 Performance Evaluation

In this section, we present the results that we obtained after applying the optimization strategies (the changes described in Subsection 3.1 apply for compressed SIKEp610 only). We performed our experiments and report the results in Table 3, targeting the processor Cortex-M4 using the board STM32F407VG and the benchmark framework PQM4 [20], running it @24MHz which sets the processor to zero wait state, and @168MHz, providing the performance cost for systems, running at the maximum controller speed, showing the protocol execution time in seconds. We have placed our code segment in the ROM memory while the data segment is placed in the RAM region, where the use of the CCM RAM is dedicated to the storage of large local data structures that exceed the size of the stack. The obtained performance improvements are a result of the

Table 3. Compressed SIKE Round 3 performance cost and speedup @24MHz aiming precision of the reported clock cycles and @168MHz reporting timing results for real world scenario.

Implementation	Timing [cc×10^6] @24MHz				Speedup	Timing Total @168MHz	
	KeyGen	Encaps	Decaps	Total	[%]	[cc×10^6]	[sec]
`SIKEp434 compressed`							
SIDH v3.3[1]	1,088	1,715	1,272	2,987	94.1	3,017	18.00
Seo et al.[2]	79	133	98	232	24.9	240	1.43
Anastasova et al.[3]	76	119	89	209	16.7	246	1.46
This work	68	99	74	174	-	212	1.26
`SIKEp503 compressed`							
SIDH v3.3[1]	1,638	2,601	1,920	4,521	94.5	4,519	27.00
Seo et al.[2]	111	181	137	319	21.8	333	1.98
Anastasova et al.[3]	99	164	125	289	13.9	359	2.14
This work	89	143	106	249	-	318	1.89
`SIKEp610 compressed`							
SIDH v3.3[1]	3,244	4,909	3,889	8,798	94.5	8,775	52.00
Seo et al.[2]	220	333	278	611	20.4	627	3.73
Anastasova et al.[3]	191	306	255	561	13.2	635	3.78
This work	187	267	219	487	-	564	3.36

[1] [35], [2] [12], [3] [11]

integration of the target-specific subroutine implementation since the slow frequency used for the precise measurements ensures zero wait states thus the use of the CCM RAM has no impact on the overall execution time. The reported results show the performance of compressed SIKE Round 3, where the work presented in [5] and [6] is adapted to the implementation design.

We compare the results with the previous best-reported implementation strategies of the finite field arithmetic needed for the long integer operations in SIKE after integrating them into the compressed SIKE protocol. Even though we propose the first integration of compressed SIKE on low-end Cortex-M4, we integrate the code presented in [12] and [11] to provide comparison results. Table 3 reports performance optimizations of 25%, 21% and 20% for SIKEp434, SIKEp503 and SIKEp610, respectively, comparing our work with [12] and 16%, 14% and 13%, compared to the design in [11]. The measurements when running the board @24MHz represent a precise number of clock cycles, however, for real-time applications, where the execution speed is of great importance, we increase the frequency to 168MHz, offered by the STM32F407-Discovery board, and report the performance in terms of clock cycles and seconds. We observe that the execution of the compressed SIKE mechanism is significantly improved, where it still represents its main drawback, while offering the smallest key sizes, and reducing the communication latency to minimal in comparison to the rest of the post-quantum candidates.

Table 4. PQC Round 3 finalists and alternate candidates timing results, memory usage and transmitted data on STM32F407VG using PQM4 [20].

Implementation	Timing [cc$\times 10^6$]			Memory [B]			Data [B]
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	pk+ct
			Security Level I				
Kyber512	0.46	0.57	0.53	2,396	2,484	2,500	1,568
ntruhps2048509	79.66	0.56	0.54	21,392	14,068	14,800	1,398
lightsaber	0.36	0.49	0.46	5,332	5,292	5,308	1,408
BIKE L1	25.06	3.40	54.79	44,108	32,156	91,400	3,113
FrodoKEM640aes	48.35	47.13	46.59	31,992	62,488	83,104	19,336
FrodoKEM640shake	79.33	79.70	79.15	26,600	51,976	72,592	19,336
SIKEp434compressed	68.26	99.50	74.86	68,636	41,380	7,940	433
	65.4%	47.6%	3.9%				56.1%
SIKEp434	41.28	67.40	72.02	6,108	6,468	6,748	676
			Security Level II				
SIKEp503compressed	89.76	143.17	106.26	88,192	53,696	9,008	505
	54.4%	49.9%	4.5%				54.5%
SIKEp503	58.12	95.53	101.73	7,360	7,736	8,112	780
			Security Level III				
Kyber768	0.76	0.92	0.86	3,276	2,968	2,988	2,272
ntruhps2048677	143.73	0.82	0.82	28,504	9,036	19,728	1,862
saber	0.66	0.84	0.79	6,364	6,316	6,332	2,080
ntruhrss701	153.10	0.38	0.87	27,560	7,400	20,552	2,276
ntrulpr761	0.74	1.29	1.39	13,168	20,000	24,032	2,206
sntrup761	10.83	0.70	0.57	61,508	13,320	16,952	2,197
SIKEp610compressed	187.67	267.35	219.80	85,740	78,532	11,524	616
	76.9%	37.2%	12.1%				53.9%
SIKEp610	106.07	194.90	196.12	10,490	10,908	11,372	948

In Table 4, we present the compressed SIKE algorithm comparing it with the rest of the post-quantum candidates in the NIST standardization effort to offer a better understanding of the advantages and disadvantages of each one of the protocols and to stress again on the main benefit of compressed SIKE – the key sizes. Even though SIKE forms part of the alternate candidates in the NIST PQ process, we believe that it is applicable in multiple scenarios, especially in low-end real-time and IoT systems, where the data transmission latency should be minimal and the resource capabilities of the devices are constrained. In Table 4, we describe the timing of the post-quantum safe candidates, the memory requirements of each scheme, and the size of the public data. We denote the Round 3 finalists in bold, where the rest of the schemes are alternate candidates. We report the timing overhead introduced by the compression mechanism of SIKE in red color and the public information overhead when running uncompressed SIKE in green.

5 Conclusions

In this work, we presented the first implementation of compressed SIKE targeting the low-end device STM32F407VG, which is the NIST recommended platform for benchmarking the post-quantum secure protocols. We increased the stack and added a new memory region placed in the CCM RAM memory, which allowed to execute compressed SIKEp610 without corrupting the memory. Further, we implement subtraction and multiplication compressed-specific subroutines in assembly code to maximize the speedup.

We hope to push SIKE and compressed SIKE further in the PQC NIST competition by continuing our optimization effort since they offer the smallest key sizes, therefore, ensure insignificant communication cost.

6 Acknowledgment

The authors would like to thank the reviewers for their detailed comments. This work is supported in parts by an award from NSF 2101085.

References

1. T. N. I. of Standards and T. (NIST)., "Post-quantum cryptography standardization, 2017-2018." https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization, last accessed on June 6, 2021.
2. D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *International Workshop on Post-Quantum Cryptography*. Springer, 2011, pp. 19–34.
3. R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, 2016, pp. 1–10.
4. C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient compression of sidh public keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 679–706.
5. M. Naehrig and J. Renes, "Dual isogenies and their application to public-key compression for isogeny-based cryptography," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2019, pp. 243–272.
6. G. Pereira, J. Doliskani, and D. Jao, "x-only point addition formula and faster compressed sike," *Journal of Cryptographic Engineering*, vol. 11, no. 1, pp. 57–69, 2021.
7. H. Fujii and D. F. Aranha, "Curve25519 for the Cortex-M4 and beyond," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2017, pp. 109–127.
8. H. Seo, "Memory efficient implementation of modular multiplication for 32-bit ARM Cortex-M4," *Applied Sciences*, vol. 10, no. 4, p. 1539, 2020.
9. M. B. Niasar, R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast, small, and area-time efficient architectures for key-exchange on curve25519," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE, 2020, pp. 72–79.

10. M. B. Niasar, R. Azarderakhsh, and M. M. Kermani, “Efficient hardware implementations for elliptic curve cryptography over curve448,” in *International Conference on Cryptology in India.* Springer, 2020, pp. 228–247.
11. M. Anastasova, R. Azarderakhsh, and M. M. Kermani, “Fast Strategies for the Implementation of SIKE Round 3 on ARM Cortex-M4.”
12. H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh, “Supersingular Isogeny Key Encapsulation (SIKE) Round 2 on ARM Cortex-M4,” *IEEE Transactions on Computers*, 2020.
13. H. Seo, Z. Liu, P. Longa, and Z. Hu, “SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–20, 2018.
14. B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani, “NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM,” in *International Conference on Cryptology and Network Security.* Springer, 2016, pp. 88–103.
15. H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, “Optimized implementation of SIKE Round 2 on 64-bit ARM Cortex-A processors,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.
16. R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, “Efficient and Fast Hardware Architectures for SIKE Round 2 on FPGA,” Cryptology ePrint Archive 2020/611, Tech. Rep., 2020.
17. B. Koziel, A.-B. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani, “SIKE’d Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020.
18. R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, “Highly optimized montgomery multiplier for SIKE primes on FPGA,” in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH).* IEEE, 2020, pp. 64–71.
19. J.-H. Phoon, W.-K. Lee, D. C.-K. Wong, W.-S. Yap, and B.-M. Goi, “Area–Time-Efficient Code-Based Postquantum Key Encapsulation Mechanism on FPGA,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 12, pp. 2672–2684, 2020.
20. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, “pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4,” 2019.
21. D. Jao and L. De Feo, “Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies,” in *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011*, 2011, pp. 19–34. [Online]. Available: https://doi.org/10.1007/978-3-642-25405-5_2
22. SIKE, “Sike website,” https://sike.org/, last accessed on June 6, 2021.
23. D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik, “Supersingular Isogeny Key Encapsulation,” Submission to the NIST Post-Quantum Standardization Project, 2017. [Online]. Available: https://sike.org/
24. P. S. Barreto, H. Y. Kim, B. Lynn, and M. Scott, “Efficient algorithms for pairing-based cryptosystems,” in *Annual international cryptology conference.* Springer, 2002, pp. 354–369.
25. S. D. Galbraith, K. Harrison, and D. Soldera, “Implementing the tate pairing,” in *International Algorithmic Number Theory Symposium.* Springer, 2002, pp. 324–337.

26. G. H. Zanon, M. A. Simplicio, G. C. Pereira, J. Doliskani, and P. S. Barreto, “Faster key compression for isogeny-based cryptosystems,” *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 688–701, 2018.
27. A. Hutchinson, K. Karabina, and G. Pereira, “Memory Optimization Techniques for Computing Discrete Logarithms in Compressed SIKE.”
28. S. Pohlig and M. Hellman, “An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.),” *IEEE Transactions on information Theory*, vol. 24, no. 1, pp. 106–110, 1978.
29. R. Earnshaw, “Procedure call standard for the ARM architecture,” *ARM Limited, October*, 2003.
30. P. Koppermann, E. Pop, J. Heyszl, and G. Sigl, “18 Seconds to Key Exchange: Limitations of Supersingular Isogeny Diffie-Hellman on Embedded Devices,” *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 932, 2018.
31. F. De Santis and G. 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*, pp. 19–21, 2016.
32. M. Hutter and E. Wenger, “Fast multi-precision multiplication for public-key cryptography on embedded microprocessors,” in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2011, pp. 459–474.
33. H. Seo and H. Kim, “Multi-precision multiplication for public-key cryptography on embedded microprocessors,” in *International Workshop on Information Security Applications.* Springer, 2012, pp. 55–67.
34. Seo, Hwajeong and Kim, Howon, “Consecutive operand-caching method for multi-precision multiplication,” *Journal of information and communication convergence engineering*, vol. 13, no. 1, pp. 27–35, 2015.
35. PQCryptov3.3, “Sidh library,” https://github.com/Microsoft/PQCrypto-SIDH.