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

Mila Anastasova, Reza Azarderakhsh, *Member, IEEE*, and Mehran Mozaffari Kermani, *Senior Member, IEEE*

***Abstract*— The Supersingular Isogeny Key Encapsulation mechanism (SIKE) is the only post-quantum key encapsulation protocol based on elliptic curves and isogeny maps between them. Despite the quantum security of the protocol, SIKE requires a greater number of clock cycles and hence does not provide competitive timing and energy consumption results. However, it is more attractive offering the smallest public key as well as ciphertext sizes, which considering the impact of the communication costs and storage of the keys could become a 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 the STM32F407VG microcontroller for benchmarking the clock cycles and on Nucleo-F411RE attached to X-NUCLEO-LPM01A (Power Shield) for measuring the energy consumption of the protocol. The low-level finite field arithmetic operations play main role in determining the efficiency of SIKE. Therefore, we mainly focus on their optimization and apply them to all NIST-required security levels. Our SIKEp434 implementation for NIST security level 1 is about 22.97% faster than the counterparts appeared in Seo *et al.* (2020), where for the SIKEp503, SIKEp610 and SIKEp751 the speedup reaches 21.10%, 19.21% and 19.08%. Finally, we benchmark energy consumption and report optimization of up to 11.9% depending on the NIST security level implementation.**

***Index Terms*— Supersingular isogeny key encapsulation (SIKE), post-quantum cryptography (PQC), ARM Cortex-M4.**

I. Introduction

THE increasing capabilities of quantum computers are the motivation behind post-quantum cryptography (PQC) [2]. Due to their data unit - the q-bit, and the principle of superposition, they can solve the hard mathematical problems underlying classical cryptography in a much shorter time than today's computers. Shor's algorithm [3] proves that factorization and elliptic curve discrete logarithm problems, the base of the widely used cryptosystems RSA and ECC, can be broken in polynomial time when quantum computers equipped with enough q-bits are developed, instead of exponential when classical computers are used.

Due to the rising threat of quantum computers, the National Institute of Standards and Technology (NIST) [4] initiated a standardization process for post-quantum secure algorithms in 2016. Between the years 2017 and 2020 Round 1 and Round 2 of the competition were completed and 7 finalists were announced, which are further evaluated for initial standardization, and another 8 alternate candidates that are still going through optimization process and will possibly form part of the Round 3 finalists. The last round started in the year 2020 and still assesses the candidates and their constant improvements. The final round of the competition evaluates 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), 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) [5] scheme is based on the Supersingular Isogeny Diffie-Hellman (SIDH) algorithm proposed in 2011 by Jao and De Feo [6]. 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 [7], [8]–[10] in the era of 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, is vulnerable to active attacks when one of the parties uses a

Manuscript received March 30, 2021; revised May 28, 2021 and June 19, 2021; accepted July 6, 2021. Date of publication July 27, 2021; date of current version September 30, 2021. This work was supported in part by the NSF under Award 2101085 and in part by Public Works and Government Services Canada. This article was recommended by Associate Editor W. Liu. *(Corresponding author: Reza Azarderakhs.)*

Mila Anastasova is with the Computer and Electrical Engineering and Computer Science Department, Florida Atlantic University, Boca Raton, FL 33431 USA, and also with the Institute for Sensing and Embedded Network Systems Engineering (I-SENSE), Florida Atlantic University, Boca Raton, FL 33431 USA (e-mail: manastasova2017@fau.edu).

Reza Azarderakhsh is with the Computer and Electrical Engineering and Computer Science Department, Florida Atlantic University, Boca Raton, FL 33431 USA, also with the Institute for Sensing and Embedded Network Systems Engineering (I-SENSE), Florida Atlantic University, Boca Raton, FL 33431 USA, and also with PQSecure Technologies, LLC, Boca Raton, FL 33431 USA (e-mail: razarderakhsh@fau.edu).

Mehran Mozaffari Kermani is with the Computer Science and Engineering Department. University of South Florida, Tampa, FL 33620 USA (e-mail: mehran2@usf.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2021.3096916.

Digital Object Identifier 10.1109/TCSI.2021.3096916

static key, allowing the recovery of the static key with minimal computation effort [11], [12]. The attack makes SIDH hard to adapt in the IND-CCA category, which leads to the SIKE proposal [5], introduced as an IND-CCA algorithm, applying a variant [13] of *Fujisaki-Okamoto* (*FO*) [14] transform which gives up on the static settings of the protocol to make, at best, semi-static settings possible, converting CPA-security public key encryption (PKE) into CCA-security KEM.

Since the start of the NIST standardization effort, several research groups have centered their work on the improvement of SIKE, aiming to reach efficiency in the computational time of the algorithm. In [1], [15], [16] and [17] the authors propose several strategies, targeting ARMv7-M, ARMv7-A and ARMv8 ARM-based architectures, reporting significant speedup of the algorithm timing. Several hardware implementations were proposed as well in [18], [19], [20] targeting the Xilinx Virtex 7 platform.

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

- We propose an efficient implementation design for modular addition, based on continuous alternation between addition/subtraction blocks, reducing the number of carry/borrow catchers/activators. We propose the implementation of a carry/borrow catcher/activator using a single register for both, by introducing new and reduced instruction set for storing and activating the carry/borrow flag. The newly proposed design releases one register, which permits to increase the size of the computational block. Moreover, we take advantage of the special form of the primes used in SIKE by adding a constant to the modulus value which converts it to a number ending by several all-zero words, eliminating multiple subtraction instructions per modular addition.
- We propose novel implementation designs for the multi-precision multiplication, squaring and reduction operations. We use the Floating-Point Register (FPR) set as Level 1 (L1) cache memory, usually integrated into the CPU, where we store partially computed results or operand values. This allows us to considerably reduce the expensive memory access instructions. Moreover, it allows us to introduce new designs for the before-mentioned multi-precision operations, modifying completely the instruction sequence of the inner multiplication loop, and the entire squaring and reduction execution flow, providing new and significantly more efficient implementation designs.
- 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 along with completely modifying the implementation design of the arithmetic operations, therefore, significantly decreasing the clock cycles and the energy consumption of the protocol. Using these novel strategies, we provide implementation speedup between 19% and 23% for all the NIST security levels of SIKE, where the maximum improvement is obtained for the prime SIKEp434 with 22.97% better performance compared to the counterparts in [1].

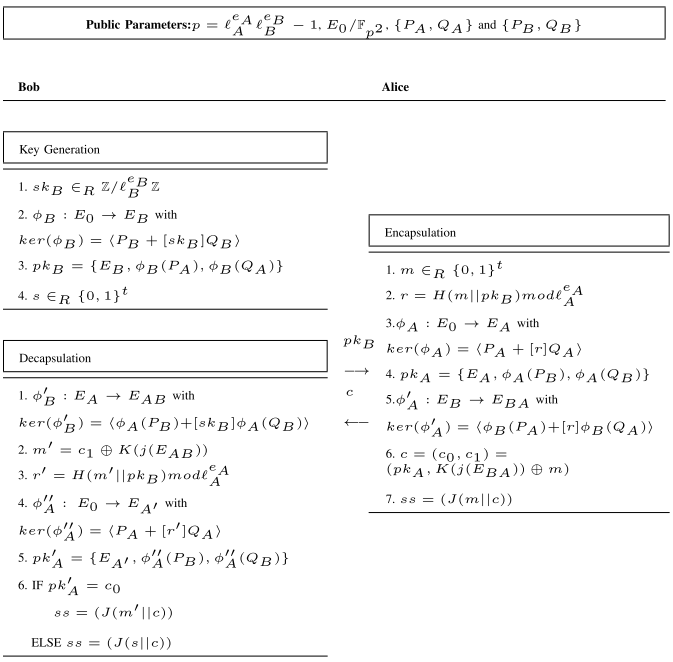

Fig. 1. SIKE algorithm [5]. H, K and J denote hash functions.

We have made our code available at the git hub repository `https://github.com/manastasova/SIKE_PhD/`.

II. Supersingular Isogeny Key Encapsulation and Target Architecture

This section presents a profound description of the steps needed for the Supersingular Isogeny Key Encapsulation mechanism. Additionally, we present an overview of the main characteristics of the target architecture of our implementation design.

A. Supersingular Isogeny Key Encapsulation

The Supersingular Isogeny Key Encapsulation mechanism was classified as part of the alternate candidates after the end of NIST Round 2 post-quantum standardization process, thus it is yet to be further optimized besides the considerable improvements so far.

Detailed graphical representation of the steps performed by both parties during the execution of the cryptography protocol is shown in Figure 1. To perform the protocol Alice and Bob start from public supersingular elliptic curve $E_0/\mathbb{F}_{p^2}$, where the prime number p has the form of $\ell_A^{e_A}\ell_B^{e_B} \cdot f \pm 1$. For efficiency purposes, the values of ℓ_A and ℓ_B are set to 2 and 3, respectively, and $f = 1$. Depending on the NIST security level of the implementation, the value of e_A and e_B vary. The basis points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ that generate $E_0[2^{e_A}]$ and $E_0[3^{e_B}]$ form part of the public parameter set.

During the key generation phase Bob forms his pair of public and private keys as a random integer (not multiple of 3^{e_B}) and a set of image elliptic curve E_B and the image points $\phi_B(P_A), \phi_B(Q_A)$, respectively.

During the key encapsulation phase Alice uses Bob's public key together with a random message m to generate her secret key as $r = H(m||pk_B)\ mod 2^{e_A}$ and applies it to the public parameters to find her secret isogeny ϕ_A and an image curve E_A, enabling her to generate her public key as $pk_A = \{E_A, \phi_A(P_B), \phi_A(Q_B)\}$. Later, she uses Bob's projection points along with her secret key r to find a second secret isogeny ϕ'_A and apply it on Bob's image curve to finally end up on E_{BA}. Alice generates a ciphertext composed of her public key and the random message m masked by the $j-$invariant of the curve E_{BA}. She finally computes the value of the shared secret as the hash of the ciphertext appended to the message m.

During the key decapsulation phase Bob 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}. He starts by revealing the value of the masked secret message m by finding the value of the masking $j-$invariant. He uses Alice's public key and his secret key to find an isogeny map ϕ'_B leading him to the image curve E_{AB}, belonging to the same isomorphic class as E_{BA}, thus featuring the same $j-$invariant. Bob reverses the XOR masking function and uses the value of m' to find Alice's secret key. He then simulates Alice's isogeny map to construct her public key, which he uses to confirm the secure communication with Alice by comparing it with her original public key. Eventually, he calculates the shared secret as $J(m'||c)$ in the case of matching public keys and using a random value, preventing further communication with Alice, in case the keys did not coincide.

B. ARMv7-M Architecture

The high demand for ARM-based devices converts it into the most widely used Reduced Instruction Set Computer (RISC) design. Featuring highly optimized power, performance, and area consumption, the ARMv7-M architecture profile is quickly integrated into the industry, forming main part of the Internet of Things world. The variety of systems based on ARMv7-M, requiring secure data transmission, instigates NIST to announce the platform Cortex-M4 as the main target for cryptographic optimizations in the scenario of low-end devices, suiting the needs of embedded and real-time systems. This work is targeting the NIST recommended microcontroller STM32F4 with an integrated Floating-Point Unit based on the FPU extension FPv4-SP.

The platform features 16 32-bit General Purpose Registers (GPRs) – `R0-R15`, where 14 of them are accessible by the programmer – `R0-R12` and `R14`. The use of the register `R13` and `R15` is reserved for the value of the Stack Pointer (SP) and the Program Counter (PC), respectively. The value of `R14` which keeps the value of the Link Register (LR) may be used when previously stored into the memory. Hence, there are 448 bits provided by the core register set, which given the size of the cryptography operands of several hundreds of bits, does not ensure the most optimal implementation, due to the requirement of repetitive load/store instructions. The Cortex-M4 instruction set requires one clock cycle per instruction except for `LDR` and `STR`, which double that number if not properly scheduled. However the nature of the post-quantum secure algorithm does not always allow to optimize the scheduling of the instructions, thus we consider the worst-case scenario where `LDR` and `STR` take 2 clock cycles.

TABLE I

ARMv7-M INSTRUCTION SET FOR MEMORY ACCESS, DATA TRANSFER BETWEEN GPR AND FPR SETS AND MAC INSTRUCTIONS. THE REQUIRED CCS PER INSTRUCTION ARE EXPRESSED IN FUNCTION OF THE NUMBER OF REGISTERS n INVOLVED IN THE INSTRUCTION. FOR MORE DETAILS REFER TO [22]

Instruction	Functionality	Timing (CC)
(V) LDR	$R_n \leftarrow$memory	2
	$S_n \leftarrow$memory	
(V) LDM	$R_n - R_m \leftarrow$memory	1+n
	$S_n - S_m \leftarrow$memory	
(V) STR	memory$\leftarrow R_n$	2
	memory$\leftarrow S_n$	
(V) STM	memory$\leftarrow R_n - R_m$	1+n
	memory$\leftarrow S_n - S_m$	
VMOV	$R_n \leftarrow S_m$	1
	$S_n \leftarrow R_m$	
UMULL	$Rd_1, Rd_2 \leftarrow R_n \times R_m$	1
UMAAL	$Rd_1, Rd_2 \leftarrow R_n \times R_m + Rd_1 + Rd_2$	1
UMLAL	$Rd_1, Rd_2 \leftarrow R_n \times R_m + Rd_1, Rd_2$	1

The platform offers a larger set of Floating-Point Registers (FPRs) which is entirely accessible by the programmer. The 32 32-bit registers `S0-S31` ensure another 1024 bits which, if used as a L1 cache, require a single clock cycle per 32-bit data access [21]. The data transfer between GPRs and FPRs promises instant information retrieval, replacing the slow memory accesses by data shift among different register types, using the stall-free `VMOV` instruction. Table I shows in detail the memory access and data relocation instructions used in this work together with the number of clock cycles per instruction.

Besides the load-store and move instructions, used to temporarily store data, the ARMv7-M architecture features the extremely optimal Multiply ACcumulate (MAC) instructions, which perform several mathematical operations in a single clock cycle. Even more, the size of the destination value length is doubled by using two, instead of only one, GPRs, storing the least and most significant 32-bits of the result, respectively. Table I shows a detailed view of the MAC instructions and their functionality.

III. PROPOSED FINITE FIELD ARITHMETIC COMPUTATIONS

The arithmetic operations required for the execution of SIKE are in a pyramid-like structure, where the topmost level comprises SIKE complex isogeny computations, whereas the bottom-most is composed of finite field arithmetic operations, whose optimization ensures impact on the overall performance of the cryptosystem.

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

A. Modular Field Addition

Modular addition consists of adding two operands `A` and `B` and reducing the result modulo `p` supposing it

Algorithm 1 Modular Addition Algorithm With 4 Word Operands, Presenting the Carry/Borrow Propagation When Applying the Add/Sub Block Alternation Technique

1. (borrow0, T0) = A0 − p0
2. Result0 = T0 + B0
3. (carry0, T1) = A1 + B1
4. Result1 = T1 − p1 − borrow0
5. (borrow1, T2) = A2 − p2
6. Result2 = T2 + B2 + carry0
7. T3 = A3 + B3
8. (borrow2, Result3) = T3 − p3 − borrow1

exceeds 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 (((A + B) − p) +(mask&p)). Supposing that the value of A + B exceeds the finite field, it should be brought back into $\mathbb{F}_p$ by subtracting p. Thus, the final addition of p should be revoked which is achieved by equaling mask = 0×0. Otherwise, (A + B) − p will result in a negative value and the subtracted p should be added back by making mask = 0×FFFFFFFF. This masking strategy ensures the constant time performance of the arithmetic operation.

The viability of modular addition of long integers into cryptography instigates researchers to continuously propose optimizations. In [23] the authors provide an efficient assembly implementation of the modular addition and subtraction, replacing the previous portable, however, slow C code implementation, and present an overall speedup of SIKE. Further, in [1] the authors suggest a completely new design, introducing the idea of operand blocks, each one consisting of four 32-bit words. Using this technique, A + B − p is performed by applying addition followed by subtraction to each 128-bit block consecutively. The carry/borrow flag values produced after each block operation is kept into (GPRs) so that it can be later propagated to the following block, exploiting a novel carry/borrow catcher/activator technique. In this work, we propose three novel ideas for outperforming the previous implementations of modular addition, while we keep the idea of operand blocking.

First, we propose to alternate the sequence of additions and subtractions through the operand blocks. This novel technique decreases the number of carry/borrow catcher/activator operations. In particular, the implementation of A + B − p, supposing A and B consist of four blocks each, where we denote the least significant blocks as A0, B0 and the most significant as A3, B3, is described in Algorithm 1. The consecutive execution of line 2, 3 and 6, 7 eliminates the need of carry catcher and activator since the ADC(S) instructions ensure the carry flag propagation, similarly to the consecutive execution of line 4, 5 eliminating the need of borrow catcher and activator using SBC(S). Figure 2 graphically illustrates the carry/borrow propagation between consecutive add/sub blocks,

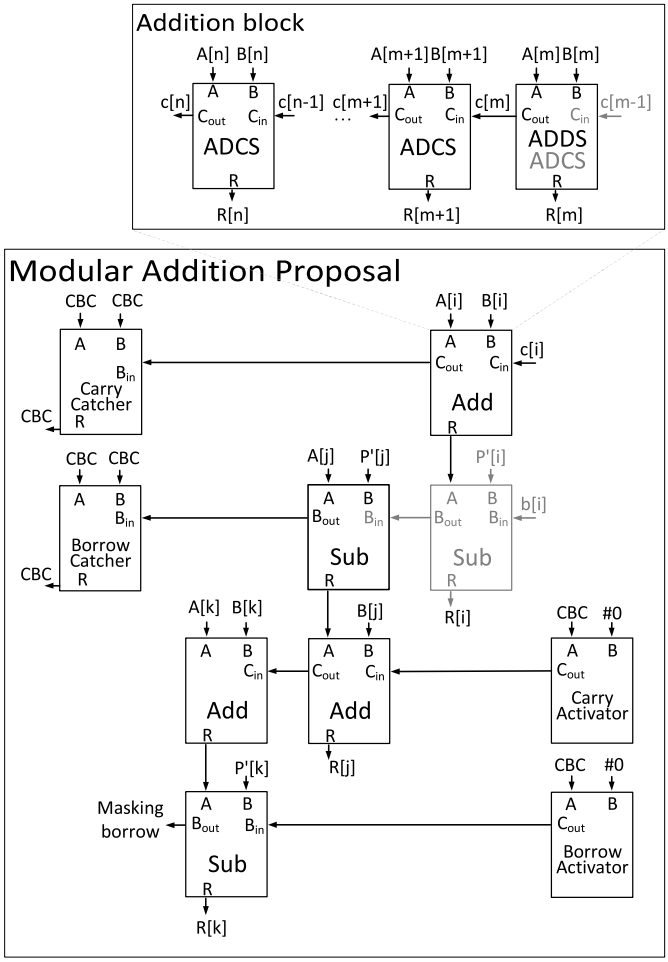

Fig. 2. Proposed modular addition design with optimized Carry/Borrow Catcher (CBC) by an alternating sequence of add/sub blocks. The k least significant subtractions (k depends on the prime), marked in gray, are eliminated by replacing P = $2p$ with P′ = $2p + 2$.

where we avoid the use of carry/borrow catcher/activator. It is important to note that, despite the alternation of the addition/subtraction operations, the last performed instruction should be always a subtraction, since the final borrow determines the mask value for the last addition with mask&p.

The block operation alternations, in addition, allow freeing one extra register, which we use to increase the size of the block from 4 to 5 (or 6 if part of the most significant block) 32-bit values, therefore, to reduce the number of blocks. Figure 2 illustrates the modular addition performed for p503 when our design is applied, where only 3 addition/subtraction blocks are needed, instead of 4 needed in [1]. Our design allows exploiting the advantage of consecutive memory accessing using LDM which requires $N + 1$ clock cycles, with N the number of loaded words, reducing the overhead clock cycles from 4 to 3. Even though insignificant to the modular addition, the invocation ratio of the operation turns to have an effect on the overall SIKE execution time.

Second, we propose new instruction set for the implementation of the carry/borrow catcher/activator. We use the SBC and RSBC instructions for carry/borrow catcher and activator, respectively. For the implementation of the former, we subtract

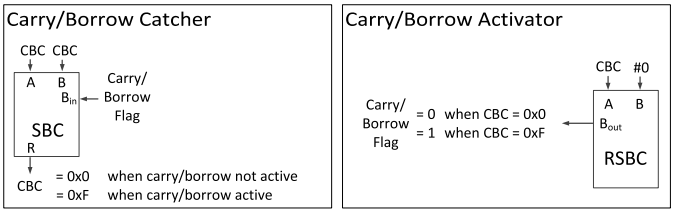

Fig. 3. Optimized instruction set for carry/borrow catcher/activator using SBC and RSBC instructions.

the carry/borrow catcher register (CBC) registers from itself with borrow propagation activated. Therefore, in case the flag was active the result will be $0\times$FFFFFFFF, otherwise, will equal 0×0. For the implementation of the latter, we subtract this value from 0×0, where the flag will be activated if the value of the carry/borrow catcher register was $0\times$FFFFFFFF and deactivated if it was 0×0. Both, the carry/borrow catcher and activator require a single instruction, therefore, contribute to the speedup of the modular addition. The implementation is presented graphically in Figure 3. Furthermore, we applied the reduced carry/borrow catcher/activator instruction set to the modular subtraction where we also report performance improvement.

Third, we propose even further optimization for the modular addition by considering the special form of the prime numbers used in SIKE. The prime p has the form of $2^{e_A}3^{e_B}-1$ where, due to the value of the exponents, it forms a number with multiple 32−bit all-one least significant words. Furthermore, the modular reduction value in SIKE is $\mathtt{P} = 2p$ (with the purpose of saving the last subtraction when Montgomery reduction is performed), where the $\times 2$ operation simply results in shifting the number 1 bit to the left. Therefore, the last words of P consist of multiple $1's$ and a 0 at the end. We noticed that if we add 2 to the value of P, $\mathtt{P'} = \mathtt{P} + 2$, we obtain multiple all-zero least significant words. Thus, when performing (A + B) − P′ several SBC(S) instructions may be skipped, significantly reducing the number of instructions per modular addition. We apply this strategy by performing the following steps:

1) T = (A + B) − P′
2) T + ((MASK&P) + 2)

The proposed design is graphically represented in Figure 2, where the least significant k skipped subtractions are marked by gray color, where k depends on the SIKE security level implementation. For step 2) the value of P′ should be added back to the partial result supposing T is negative, otherwise we need to add back the value of 2 that we have added to P. By first masking P, we obtain P or 0×0 to which we accumulate the value of 2 to get the final result.

Our three novel proposals for the implementation of modular addition, which we apply to the subtraction when possible, result in outperformance of the previous development designs and significantly improve the overall execution time of SIKE. Moreover, since the proposed implementation is completely scalable, it is adapted to all the four SIKE primes. The results obtained are shown in Table II in clock cycles, where we show the improvement in percentage. We report 28.46%, 27.37%, 28.4% and 28.42% of speedup, respectively for p434, p503, p610 and p751 compared to the previous best-reported results. We applied the optimizations to the modular subtraction where we obtained up to 19.18% speedup for the different SIKE security implementations.

TABLE II
COMPARISON BETWEEN THE SIKE FINITE FIELD ARITHMETIC OPERATIONS MEASURED ON STM32F407VG CPU@24MHz

Implementation	Lang	Timing [CC] Speedup[%]							
		$\mathbb{F}_p$add		$\mathbb{F}_p$sub		$\mathbb{F}_p$add		$\mathbb{F}_p$sub	
		CC	%	CC	%	CC	%	CC	%
		SIKEp434				SIKEp503			
SIDH v3.3[1]	C	932	80.58	519	66.86	1,024	80.57	609	68.80
Seo et al.[2]	ASM	254	28.74	208	17.31	275	27.64	223	14.80
Seo et al.[3]	ASM	253	28.46	207	16.91	274	27.37	227	16.30
This work	ASM	181	-	172	-	199	-	190	-
		SIKEp610				SIKEp751			
SIDH v3.3[1]	C	1,314	81.96	836	72.73	1,570	82.36	996	74.20
Seo et al.[2]	ASM	-	-	-	-	388	28.61	284	9.51
Seo et al.[3]	ASM	331	28.40	272	16.18	387	28.42	318	19.18
This work	ASM	237	-	228	-	277	-	257	-

The referred results are presented in: [1] [24], [2] [25], [3] [1]

B. Multi-Precision Multiplication, Squaring, and Reduction

The friendly form of the SIKE primes allows the use of Montgomery multiplication, which performs the multiplication and the reduction, in an efficient and optimized way. It consists of two multi-precision multiplications and addition, where the sequence of instructions and the reduction of memory accesses play crucial role in optimizing the execution time. The mirror shape of the squaring allows even further improvements of the design which was first suggested in [1].

In this section we propose novel implementation strategies for minimizing the number of clock cycles required for the multi-precision multiplication, squaring and reduction functions. The scalability of our design allows the adaption of the implementation to different lengths; therefore, we implemented all the four different security levels of SIKE and observed a significant speedup for all of them.

1) Multi-Precision Multiplication: The multi-precision multiplication invocation rate inside the cryptographic protocol is distinguishable high since the computationally expensive operations (i.e., inversion) are replaced by several multiplications. As a result, several research groups focus on optimization strategies lessening the execution time of the subroutine.

The implementation shown in [23] is based on the Karatsuba multiplication [26], with time complexity of $\mathcal{O}(n^{log_2 3})$, where instead of performing one multiplication of operand sizes n, three half-size multiplications are completed together with several additions/subtractions. Furthermore, in [27] the authors implement a 2-level Karatsuba multiplication resulting in several 64×64-bit multiplications instead of one 256×256-bit operation. Later, given the architecture of the target processor, the use of the low-cost Multiply ACcumulate (MAC) instructions is proposed in [28] and is integrated into the multi-precision multiplication design. The use of MAC instructions, combined with a reduced number of memory accesses, leads to the most efficient implementation algorithms for the target

TABLE III

INSTRUCTIONS AND CLOCK CYCLES FOR MEMORY ACCESS AND REGISTER MOVE OPERATIONS, CONSIDERING DIFFERENT IMPLEMENTATION DESIGNS – OPERAND SCANNING (OS), OPERAND CACHING (OC) AND REFINED-OPERAND CACHING (R-OC). FOR (V)LDM AND (V)STM THE NUMBER OF ACCESSED REGISTERS IS SHOWN IN PARENTHESES

Memory accesses								
Design	SIKEp434				SIKEp503			
	LDR	STR	VMOV	Total [CC]	(V)LDR (V)LDM	STR	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	34 + 2(×16)	34	140	310
	SIKEp610				SIKEp751			
	LDR	STR	VMOV	Total[CC]	LDR	STR	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	127	12	190	468	198	32	240	700

platform ARM Cortex-M4, named Operand Caching (OC) [28] 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, where the size of the row is defined as the number of consecutive accumulative multiplications performed per column. Further improvements of the algorithm are presented by Seo and Kim [29] and *et al.* [30], where the Consecutive Operand Caching and the Full Operand Caching implementations are proposed, aiming to further optimize the memory accesses by re-configuring the instruction flow.

In [7] 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 set of MAC instruction is also considered in [31], 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 [25] the authors propose an optimized strategy, integrating the instruction UMULL, which handles the initialization of the register while the 32×32-bit accumulative multiplication is performed. Later, the design is even further improved in [1], where the authors implement an efficient multiplication strategy for all SIKE primes. They propose novel management of the register set for caching four words per operand, naming it Refined-Operand Caching (R-OC) where they increase the size of the rows in comparison to the previous OC implementations.

This work integrates the FPR set to store the partial results or the value of the operands, depending on the length of the prime number used in a given SIKE security level. The idea for using FPRs was first integrated into the context of post-quantum cryptography by Alkim *et al.* in [32], however, was focused on NTRU lattice-base polynomial multiplication. We apply the use of FPRs inside the context of SIKE and the multi-precision multiplication, squaring and reduction, which allowed the implementation of a new design of the arithmetic operations, reordering the instruction flow and changing the execution pattern.

For representing the multiplication we use rhombus notation as shown in Figure 4, 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 $k \in \{n-1, \ldots, 2, 1, 0\}$ with 0 being the least significant 32-bit word. The number of limbs n varies based on the number of bits needed to represent an 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]) = A · 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.

In this work, we use of FPR set for the storage of partial results or the operand values, which allows new combinatorics solution to the sequence of performed operations during the multiplication, squaring and reduction and results in decreased execution time of the algorithm. Based on the length of the operands, however, we use two distinct multi-precision multiplication optimization strategies, where the former introduces a completely new multiplication design and the latter keeps the instruction flow presented in [1], however integrating the use of FPRs. The changes applied, depending on the prime size of SIKE, are described as follows:

a) SIKEp503: The prime p503 requires 512 bits per operand, where the formula $n = \lfloor m/w \rfloor$ indicates the need for 16 words to keep them, consequently, both multiplication operands A and B can fit in 32 $32-bit$ registers. We have noticed that the register requirement coincides with the number of FPRs, thus we decided to keep the operand values inside the FPR set for instant access to the data where in Figure 4 we present the operand values stores into the register set in black color and the actual FPRs used to tore these values in red color. This strategy eliminates all memory accesses for loading the operands and transforms the idea of Operand Caching since we place the entire operand values into the emulated L1 cache memory region, where they are accessed by the VMOV instruction taking a single clock cycle.

The scarce GPR set of the processor ARM Cortex-M4 allows a maximum of 4 words per operand cached in the register set. Therefore, constant reload is needed, accessing the memory which introduces an overhead cycle. By storing operands into the FPR set, the access to the needed data becomes cheaper and the size of the row (the size of the inner loop of the multi-precision multiplication) can be increased without increasing the cost of data access. In the implementation design proposed in [1], each column inside a row requires one new operand limb to be loaded into the GPRs. In our design, we propose to increase the row size by re-accessing a limb and then accessing the newly required one, therefore, we access two operand limbs per partially computed column. Even though we double the number of accessed 32-bit words, the cost per partial column remains the same since the data is previously stored in the emulated L1 memory, replacing the

Algorithm 2 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} //S0-S15 <- A0-A15
VLDM R1, {S16-S31} //S16-S31 <- B0-B15
….
UMAAL R0, R14, R2, R6 //A2*B9
UMAAL R0, R12, R3, R7 //A3*B8
UMAAL R0, R11, R4, R8 //A4*B7
VMOV R8, S22 //R6 <- B6
UMAAL R0, R10, R5, R8 //A5*B6
VMOV R8, S26 //R6 <- B10
UMAAL R0, R9, R1, R8 //A1*B10
STR R0, [SP, #4*11]
LDR R0, [SP, #4*12]
UMAAL R0, R14, R2, R8 //A2*B10
UMAAL R0, R12, R3, R6 //A3*B9
UMAAL R0, R11, R4, R7 //A4*B8
VMOV R7, S23 //R7 <- B7
UMAAL R0, R10, R5, R7 //A5*B7
VMOV R7, S27 //R7 <- B11
UMAAL R0, R9, R1, R7 //A1*B11
….
```

load instruction with a data transfer (move) between the two different register sets.

Algorithm 2 shows the instruction flow managing the previously displaced data re-access and the access to the new 32-bit word. We reserve 5 GPRs for the operand A, therefore, we only have 3 left for the operand B. For row size of 5, we need to perform 5 accumulative multiplications among 5 words from both operands and store the partial result. To obtain the missing 2 words from B we constantly switch the GPR values, where the access of the newly needed word of B replaces a value that should be re-accessed in the next column computation. Thus, each iteration of the inner multiplication loop requires to access one new limb of B and to re-access the previously displaced one. This imposes a constant of 2 CCs per column for data accessing when using the VMOV instruction. Therefore, our new 5×5-limb multiplication referred as multiplication p503 in Figure 4 has the same cost as the 4×4 multiplication, proposed in [1], where the LDR instruction is used.

The optimization observed is a result of the increased row size, thus, decreased number of rows, leading to fewer partial results which have to be stored/loaded to/from the stack due to the processor architecture and the limited number of registers. Figure 4 shows the implementation of our new implementation strategy, where the row size is increased from 4 to 5, therefore ends up with only 3 rows and therefore 2 partial results – after row number one and after row number 2. Thus, we reduce the number of accesses to the stack. The proposed implementation significantly outperforms the previous designs.

b) SIKEp434, SIKEp610 and SIKEp751: Given the length of the prime numbers p434, p610 and p751, the operands are either too short or too long to fit inside the FPR set, which imposes either unused registers or lack

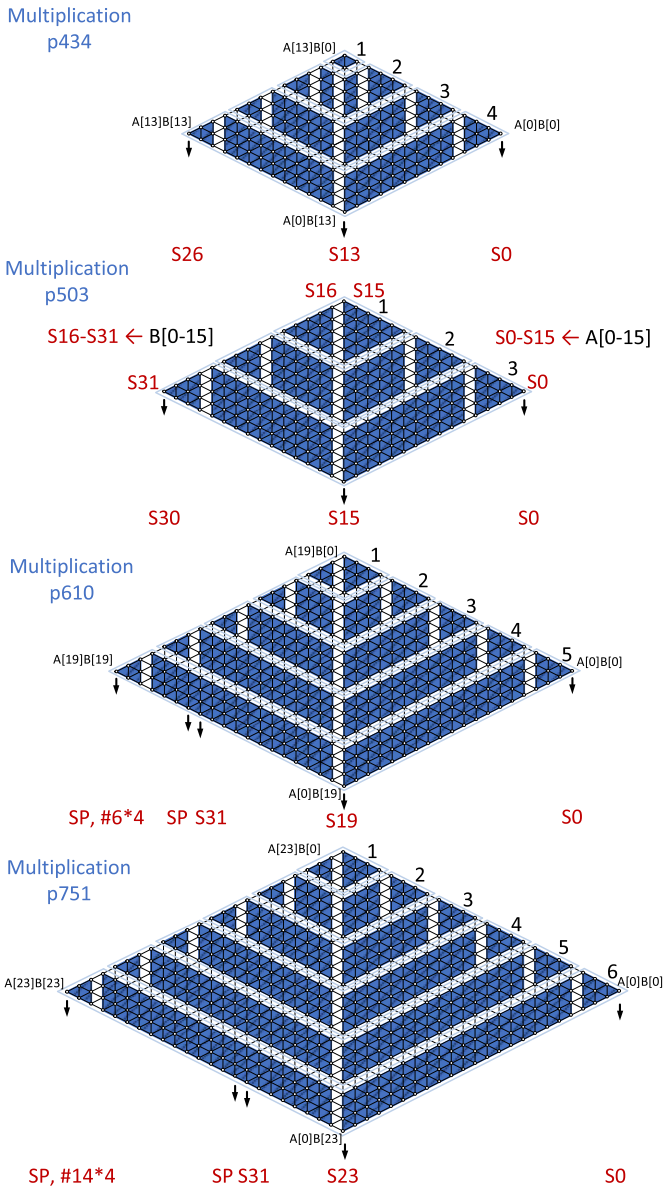

Fig. 4. Rhombus representation of SIKE multiplication for all four primes. The implementations of p434, p610 and p751 use FPR set as L1 cache for the partial results, avoiding the use of the stack. The p503 multiplier uses the FPR set to store the operands A and B and hence reduces the cost of accesses to their limbs, resulting in reduction of the number of rows.

of them where additional access to the memory is required. Therefore, for the multiplication implementation of the given primes, we decided to use the FPR set as a L1 cache, storing the partial values, produced after the row computation.

Depending on the resulting size, where the number of FPRs is not enough to store the result, the stack is used to store the last 8 and 16 limbs, for p610 and p751, respectively.

The length of p434 requires $n = \lfloor m/w \rfloor = \lfloor 434/32 \rfloor = 14$ words to store each operand and using maximum row size of 4×4 results in 4 rows – 1 of size 2×2 and 3 of size 4×4. Therefore, the idea of increasing the size of the rows and decreasing their number would eliminate only the first-row partial result which is of length 4 and will have an insignificant

impact of only 8 clock cycles. Thus, we use the emulated cache for the partial result, where we noticed that the multi-precision multiplication result is a 28-word value with 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 so that we completely eliminate the stack accesses.

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, therefore we are not able to eliminate all `LDR` instructions that access the operands and thus would not obtain the best performance when loading the operands into the FPRs. For the large primes p610 and p751 we use the FPR set to store the partial result, similarly to 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 propose a solution that uses the FPRs for the storage of the partial results of the first four rows for both primes and then uses the stack for the last 8 or 16 words, computed in the last row(s), as presented in Figure 4. 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 $n = \lfloor m/\omega \rfloor$, which optimizes the stack usage, since the following reduction uses the least significant n words as an operand for the second multiplication operation and these limbs should be loaded much more often, while the n most significant words are accessed fewer times.

2) Multi-Precision Squaring: Squaring is a special case of multiplication where the operands 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.

The previous squaring strategies include the Scott and Szczechowiak's [33] implementation, which applies Operand Scanning multiplication technique to the inner squaring loop dividing the procedure into 2 blocks – one performing only doubled multiplications and another performing doubled multiplication accumulated to a square product in each column of operation. Later, in [34] the design is improved by separating the execution of the latter block type – square accumulation is performed after the multiplication is completed. Later, in [35] *Seo et al.* introduce the Sliding Block Doubling (SBD), replacing the underlying multiplication by Product Scanning and eliminating the operand doubling by left shifting. Further improvement is introduced in [7] by applying the Operand Caching strategy to the inner multiplication loop, therefore reducing the register pressure of the former implementation designs. In [31] the authors propose a new execution sequence, which is particular for 256−bit integers, therefore, cannot be

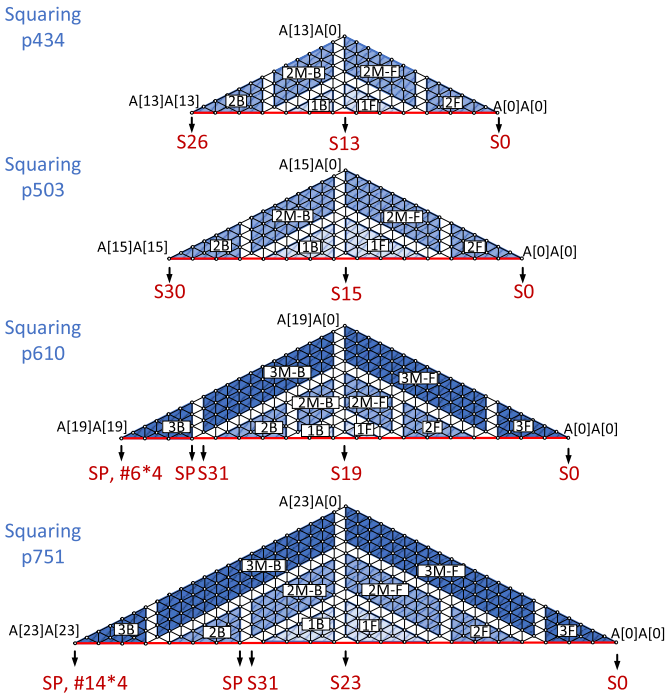

Fig. 5. Squaring implementations for all the SIKE primes. The sub-squaring blocks are denoted with i**F** and i**B** and the sub-multiplications blocks – with i**M-F** and i**M-B**.

adopted to the SIKE operand sizes. In [1] the authors propose an efficient implementation, defining 2 block types – sub-multiplication type, following their Refined-Operand Caching multiplication optimizations and sub-squaring type, where they use the SBD technique. Moreover, they precompute the double of the operand and use the stack to save its value.

In this work, we propose a new implementation strategy for multi-precision squaring which reorders the instructions to optimize the memory accesses. Our design, similar to the proposed multiplication, uses the FPR set to store the partial values after the computation of each row. Moreover, we load words of operand A in the FPR set like the p503 multiplication strategy, which saves several memory accesses for the load of the operand and reduces the stack usage.

We separate our implementation into 2 different block types – sub-squaring and sub-multiplication type similar to [1]. We perform the sub-squaring implementation block at the beginning and at the end of each row, where its middle part consists of sub-multiplication blocks. In Figure 5 we represent the sub-squaring blocks as i**F**ront (i**F**) and i**B**ack (i**B**), with i being the number of the row. We denote the sub-multiplication block, which follows the multiplication technique, described in [1] and referred to as a Refined-Operand Caching (R-OC), as i**M**iddle-**F**ront (i**M-F**) and i**M**iddle-**B**ack (i**M-B**). These rows are not computing words with equivalent indexes, therefore, the only difference from the multiplication R-OC technique is the operand doubling. However, since the doubled values in i**M-B** are reused from the i**M-F** block, the $\times 2$ operations in i**M-B** there are avoided, optimizing the implementation.

Our main contribution related to the squaring function is the new design of the row shapes and the reordering of the

execution flow. We start our implementation from the bottom of the half-rhombus, as shown in Figure 5. Every row starts with a sub-squaring block, which like the design in [33] accumulates the squared values with the multiplication product column-by-column. We first calculate the doubled operand $2 \times A[i]$, then the multiplication of the limbs $(2 \times A[i]) \times A[j]$ and last compute the value of the squaring $A[i] \times A[i]$. We reuse the doubled limbs for the computation of the following columns and keep these doubled values inside the GPR set. Therefore, we need to reload the value of the limbs to use them in their non-doubled form. However, due to our implementation strategy, using the FPR set as a L1 cache, the reloading requires 1 clock cycle. We observed that, opposite to the implementation in [1], it is cheaper to load the non-doubled value inside the GPR set than to compute the doubled values, store them into the stack and later obtain them back from the memory due to the memory access instructions overhead. The sub-multiplication block implements the Refined-Operand Caching multiplication strategy. In Figure 5 the squaring implementation of all SIKE primes can be observed.

3) Modular Reduction: SIKE uses Montgomery multiplication since the reduction step takes advantage of the Montgomery-friendly prime numbers, used for the four different NIST security levels. The operation is impacted by the techniques used in the design of the multi-precision multiplication since the reduction 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 and accumulation are skipped, since they consist of $\times 0$ multiplications, with $k = \{6, 7, 9, 11\}$ for NIST security levels 1, 2, 3, and 5, respectively, pointed out by *Costello et al.* in [36]. The Montgomery reduction algorithm is described in Algorithm 3, where the Montgomery-friendly SIKE primes make $P' = 1$, the division results – right shifting, since R is power of 2 and the last subtraction – omitted, since SIKE is implemented in $mod\ 2p$, specifically to avoid the given operation.

Optimizations considering the design of the reduction implementation were proposed in [37] by the Hybrid Montgomery multiplication method, targeting small 8-bit AVR microcontroller, showing significant improvement in the timing results. In [15], *Seo et al.* proposed even further acceleration of the Hybrid Scanning method, benefiting from the MAC instructions in the SISD implementation of the algorithm. In [1], the authors propose further improvements, increasing the row size up to 4, similar to the multiplication, where they report performance records.

In this work, we propose a novel implementation of the reduction algorithm, which optimizes the number of rows and, therefore, significantly reduces the number of memory accesses. Our reorganization of the sequence of instructions along with the usage of the FPR set considerably outperforms the previous best implementation. In continuation, we describe the two optimization strategies that allowed us to retrieve the mentioned results.

First, we apply a novel instruction flow design shown in Figure 6. Our proposal approaches the multiplication $M \cdot Q$ starting from the least significant non-zero values of M and

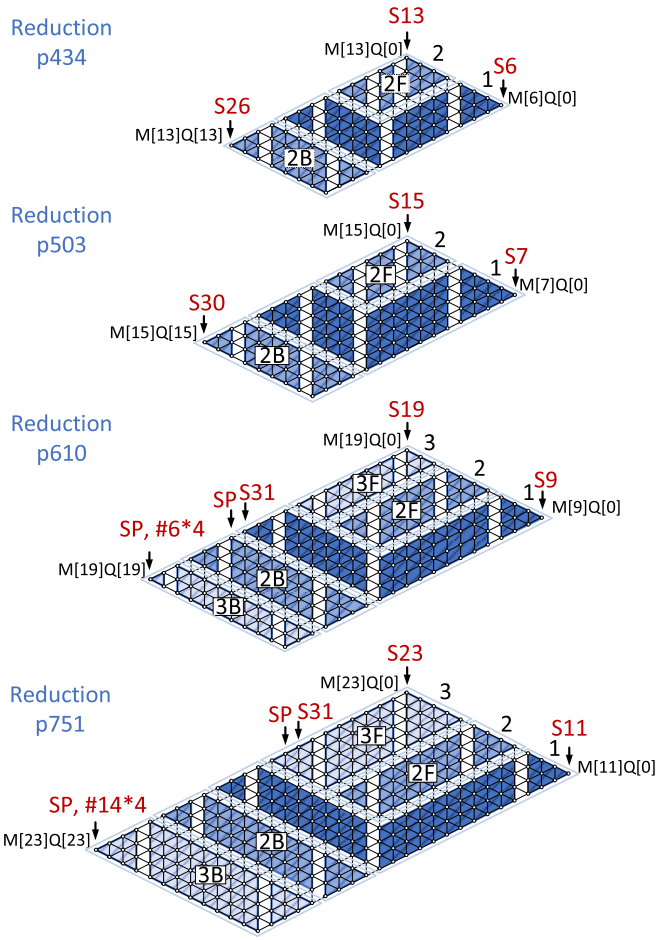

Fig. 6. Reduction for the primes p434, p503, p610 and p751. The front part of a row calculation is denoted by i**F** and the back with i**B**.

Algorithm 3 Montgomery Multiplication [38]

INPUT: M, R, M' = $-M^{-1}$ (mod R), A, B
OUTPUT: A · B · R^{-1} (mod M)
1. T = A · B
2. Q = T · M' (mod R)
3. T = (T + Q · M) / R
4. **IF** (T > Q) **RETURN** T − M
5. **RETURN** T

multiplies them with the values of Q, where M is the modulus value and Q denotes the least significant n words from the result $T = A \cdot B$ as shown in Algorithm 3. 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 the following multiplications. Therefore, the values of $Q[0]$-$Q[k]$ can be directly used due to the M 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. Therefore, we reconfigured the execution flow of the rows by modifying the direction of computation. Starting with the first row, shown in Figure 6,

TABLE IV
COMPARISON BETWEEN THE SIKE FINITE FIELD ARITHMETIC OPERATIONS MEASURED ON STM32F407

Implementation	Timing [CC] Speedup[%]							
	$\mathbb{F}_p$mul		$\mathbb{F}_p$sqr		$\mathbb{F}_p$mul		$\mathbb{F}_p$sqr	
	CC	%	CC	%	CC	%	CC	%
	SIKEp434				SIKEp503			
SIDH v3.3[1]	17,964	95.72	17,964	96.69	23,364	95.93	23,364	96.86
Seo et al.[2]	1,110	30.72	981	39.45	1,333	28.58	1,139	35.56
Seo et al.[3]	1,011	23.94	889	33.18	1,221	22.03	1,024	28.32
This work	769	-	594	-	952	-	734	
	SIKEp610				SIKEp751			
SIDH v3.3[1]	35,047	95.70	35,047	96.66	49,722	95.77	49,722	96.90
Seo et al.[2]	-	-	-	-	2,744	23.36	2,242	31.18
Seo et al.[3]	1,869	19.42	1,535	23.71	2,577	18.39	2,066	25.31
This work	1506	-	1171	-	2103	-	1543	-

The referred results are presented in: [1] [24], [2] [25], [3] [1]

we use m words from M and load words from Q until we reach the $(k+m)^{th}$ limb. After we reach $T[k+m]$, we change the direction of the execution flow and start loading words from M, while we reuse the last loaded m words from Q. The following rows implement the same idea, where although it appears that the rows are interrupted in the middle, the same row size is conserved as well as the same number of operand words are required and loaded. We mark these rows as $i\mathbf{F}$ and $i\mathbf{B}$ in Figure 6, where i is the number of the row, where the tail $i\mathbf{F}$ and the beginning $i\mathbf{B}$ form m accumulative 32×32-bit multiplications. Implementing the proposed strategy we decrease 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 which significantly optimizes the design.

Second, we exploit the FPR set similarly to the multiplication. We reduce the row size if needed by using the described technique in Section III-B.1, where 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 in Figure 6. Furthermore, since the Montgomery reduction requires the accumulation of the $Q \cdot M$ multiplication result to the value of the temporary $T = A \cdot B$ multiplication, we carefully choose where to store the value of T. 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 required only for accumulating the result. To increase the performance result we have placed the least significant value of T into the FPRs which ensures one clock cycle per `VMOV` instruction, 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 the obtained number of clock cycles per modular multiplication and squaring and report the percentage improvement in Table IV. We observe 23.94%, 22.03%, 19.42% and 18.39% of speedup for the modular multiplication and 33.18%, 28.32%, 23.41% and 25.31% of improvement for the modular squaring operations for SIKEp434, SIKEp503, SIKEp610 and SIKEp751, respectively. It should be noted that the improvement of the multi-precision operations is impacted by the length of the operands since it determines the size and the length of the rows that can be formed in the multiplication, squaring and reduction functions. Thus, the most significant speedup is observed for the primes which allow the most maximized partition of the rows for all the multi-precision operations.

IV. PERFORMANCE EVALUATION

In this section, we present the results that we obtained after applying the proposed optimization strategies. We performed our experiments, targeting the processor Cortex-M4 using the boards STM32F407 Discovery Kit, recommended by NIST as a low-end device, for benchmarking the clock cycles and the memory usage. We use the benchmarking framework pqm4 [39], running it @24MHz, which sets the processor to zero wait state, eliminating the instruction fetching stalls. We used the NUCLEO-F411RE and X-NUCLEO-LPM01A for measuring the energy consumption, basing our experiment on the PQPS [40] benchmarking framework. We present the first work of the NIST PQC standardization process Round 3 on SIKE, thus the comparison includes SIKE Round 2 performance. However, the similarities between both SIKE rounds implementations ensure no impact on the performance.

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 II and Table IV we present the clock cycles required for the execution of each one of the finite field operations 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 improvements show the importance of careful instruction management when hand-coded assembly implementation is designed. The alternating add/sub blocks, the new carry/borrow catcher/activator reduced instruction set and the constant added to the prime resulting in all-zero least significant words resulted in considerable improvements of the execution timing of the modular addition and subtraction operations. Despite that the invocation rate of these functions cannot be compared to the multiplication and reduction impact on the protocol performance, the considerable improvements in the before-mentioned finite field operations resulted in significant overall speedup. In Table IV we show that the multi-precision multiplication outperforms the previous implementation by 18.39% up to 23.94%, and the multi-precision squaring shows up to 33.18% better results, in comparison to the best previously reported results.

Our implementations show significantly better results in comparison to the previous fastest implementation strategies [1]. In Table V, we have measured and reported the clock cycles required for the execution of the SIKE algorithm. We have obtained a speedup of 22.97%, 21.10%, 19.21% and 19.08% for the primes SIKEp434, SIKEp503, SIKEp610 and SIKEp751, respectively. Furthermore, we observe slight improvement of the memory usage reported in Table V, due to the FPR set integration into the implementation design which reduces the pressure on the stack. The

TABLE V

REPORT OF SIKE MEMORY USAGE (GREEN DENOTES MEMORY USAGE DECREASE AND RED – MEMORY USAGE INCREASE), TIMING AND SPEEDUP ON STM32F407

Implementation	Memory [B]				Timing [cc $\times 10^6$]				Speedup
	KeyGen	Encaps	Decaps	[%]	KeyGen	Encaps	Decaps	Total	[%]
SIKEp434									
SIDH v3.3[1]	6,620	6,920	7,256	7.05	650	1,065	1,136	2,202	93.67
Seo et al.[2]	6,580	6,916	7,260	7.05	74	122	130	252	44.68
Seo et al.[3]	6,188	6,516	6,860	1.50	54	87	94	181	22.97
This work	6,092	6,420	6,756	-	41	67	72	139	-
SIKEp503									
SIDH v3.3[1]	6,244	6,620	6,996	6.28	985	1,623	1,726	3,350	94.11
Seo et al.2	6,204	6,588	6,974	6.65	104	172	183	355	44.43
Seo et al.[3]	6,700	7,084	7,468	0.16	74	121	129	250	21.10
This work	6,688	7,080	7,448	-	58	96	102	197	-
SIKEp610									
SIDH v3.3[1]	9,668	10,092	10,548	5.29	1,819	3,348	3,368	6,716	94.18
Seo et al.[3]	10,244	10,668	11,140	0.07	131	241	243	484	19.21
This work	10,244	10,668	11,124	-	106	195	196	391	-
SIKEp751									
SIDH v3.3[1]	11,156	11,300	11,884	5.88	3,296	5,347	5,742	11,089	94.48
Seo et al.[2]	11,116	11,260	11,852	6.17	282	455	491	946	35.25
Seo et al.[3]	11,852	11,996	12,564	0.29	225	365	392	757	19.08
This work	11,884	12,036	12,596	-	182	295	317	613	-

The referred results are presented in: [1] [24], [2] [25], [3] [1]

TABLE VI

SIKE ENERGY CONSUMPTION MEASURED ON NUCLEO-F4 AND X-NUCLEO-LPM

Implementation	Language	Speed [MHz]	Energy [mJ]			
			KeyGen	Encaps	Decaps	Total
SIKEp434						
SIDH v3.3[1]	C	96	485.00	798.32	850.72	1,649.04
Seo et al.[2]	ASM		37.26	61.60	65.54	127.14
This work			32.96	54.16	57.85	112.01
SIKEp503						
SIDH v3.3[1]	C	96	724.96	1,198.00	1,273.00	2,471.00
Seo et al.[2]	ASM		53.03	87.89	93.55	181.44
This work			48.93	81.09	86.18	167.27
SIKEp610						
SIDH v3.3[1]	C	96	1,358.00	2,516.00	2,528.00	5,044.00
Seo et al.[2]	ASM		97.36	180.50	181.30	361.80
This work			92.03	170.13	171.15	341.28
SIKEp751						
SIDH v3.3[1]	C	96	2,435.00	3,992.00	4,273.00	8,265.00
Seo et al.[2]	ASM		172.07	280.53	301.58	582.11
This work			162.08	263.91	283.60	547.51

The referred results are presented in: [1] [24], [2] [1]

TABLE VII

STM32F407 REPORT OF PQC ROUND 3 FINALISTS AND ALTERNATE CANDIDATES TIMING RESULTS (IN CLOCK CYCLES), MEMORY USAGE AND TRANSMITTED DATA (IN BYTES)

SL	Implementation	Timing [cc $\times 10^6$]			Timing [s]	Memory [B]			Data [B]
		KeyGen	Encaps	Decaps	Total	KeyGen	Encaps	Decaps	pk+ct
Finalists									
I	mceliece348864[1]	1589.60	0.48	2.29	0.12	1,412	1,412	18,492	261,248
	Kyber512	0.46	0.57	0.53	0.05	2,396	2,484	2,500	1,568
	ntruhps2048509	79.66	0.56	0.54	0.05	21,392	14,068	14,800	1,398
	lightsaber	0.36	0.49	0.46	0.04	5,332	5,292	5,308	1,408
III	Kyber768	0.76	0.92	0.86	0.07	3,276	2,9684	2,988	2,272
	ntruhps2048677	143.73	0.82	0.82	0.07	28,504	9,036	19,728	1,862
	saber	0.66	0.84	0.79	0.07	6,364	6,316	6,332	2,080
	ntruhrss701	153.10	0.38	0.87	0.05	27,560	7,400	20,552	2,276
IV	Kyber1024	1.22	1.41	1.33	0.11	3,788	3,476	3,508	3,136
	ntruhps4096821	208.84	1.03	1.03	0.09	34,504	10,924	23,952	2,460
	firesaber	1.01	1.22	1.17	0.10	7,388	7,340	7,356	2,784
Alternate									
I	BIKE L1	25.06	3.40	54.79	2.42	44,108	32,156	91,400	3,113
	FrodoKEM640aes	48.35	47.13	46.59	3.91	31,992	62,488	83,104	19,336
	FrodoKEM640shake	79.33	79.70	79.15	6.62	26,600	51,976	72,592	19,336
	SIKEp434	**41.28**	**67.40**	**72.02**	**5.81**	**6,108**	**6,468**	**6,748**	**676**
II	**SIKEp503**	**58.12**	**95.53**	**101.73**	**8.22**	**7,360**	**7,736**	**8,112**	**780**
III	ntrulpr761	0.74	1.29	1.39	0.11	13,168	20,000	24,032	2,206
	sntrup761	10.83	0.70	0.57	0.05	61,508	13,320	16,952	2,197
	SIKEp610	**106.07**	**194.90**	**196.12**	**16.29**	**10,490**	**10,908**	**11,372**	**948**
IV	**SIKEp751**	**182.28**	**295.36**	**317.22**	**25.52**	**12,180**	**12,324**	**12,876**	**1,160**

The referred results are presented in: [1] [41]

observed performance and memory improvements are based on the optimizations of the low-level underlying finite field operations, where the speedup of the modular addition, multi-precision multiplication, squaring and reduction along with their memory usage result in an overall improvement of the protocol. The impact of the operations on the performance depends mainly on their invocation rate. The modular multiplication has the most significant effect on the algorithm execution time whereas the squaring shows less impact. However, the considerable optimization results for all the arithmetic operations compensate in the case of low invocation rate, thus, the overall performance of the protocol is a result of the cumulative effect of the execution timing of all routines. The observed performance speedup is most considerable for NIST security level 1 and slightly decreases with each following level with smallest improvement rate of 19.08% for SIKEp751. When considering the implementation design proposed in [25] we observe an improvement of 44.68%, 44.43% and 35,25% for SIKEp434, SIKEp503 and SIKEp751, respectively.

Low energy 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 measured the energy consumption using the NUCLEO-F411RE board running at 96 MHz. Table VI, reports the results we have obtained. The energy consumption is decreased with 15 mJ, 14 mJ, 20 mJ and 34mJ for the SIKEp434, SIKEp503, SIKEp610 and SIKEp751, respectively. The results correspond to 11.9%, 7.8%, 5.7% and 5.9% of improvement of the energy consumption for the four prime numbers, where it should be noted that for battery-powered devices energy consumption is the most critical parameter.

After the integration of the proposed arithmetic operations into SIKE, we have compared the NIST PQC finalists and alternate candidates in Table VII, measuring the performance of the protocols, their memory consumption and the length of the key size and ciphertext, which compose the transmitted information between the parties. The isogeny-based cryptosystem shows minimal memory usage which makes it suitable scheme for integration into resource-constrained IoT devices being the third most memory efficient protocol after the lattice-based Kyber and Saber. In terms of performance SIKE is still not comparable with these protocols even though significantly improved compared to the counterparts. However, the compact public key and ciphertext used in SIKE promise insignificant impact on communication latency, which is crucial in real-time

systems where it can become a bottleneck in the scenario of limited bandwidth.

V. 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 novel implementation strategies for the: (1) modular addition and (2) multi-precision multiplication, squaring and reduction. For (1) we take advantage of the special form of the prime number, we introduce a new instruction set for the carry/borrow catcher/activator and new add/sub block alternation technique. For (2) we integrate the use of FP register set emulating L1 cache, which allows us to introduce new designs, modifying completely the instruction sequence of the inner multiplication loop, and the entire squaring and reduction execution flow, increasing the size of the operation rows and thus decreasing their number optimizing the memory accesses.

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 a side-channel analysis of our implementations as a future project.

Acknowledgment

The authors would like to thank the reviewers for their comments.

References

[1] H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh, "Supersingular isogeny key encapsulation (SIKE) round 2 on ARM Cortex-M4," *IEEE Trans. Comput.*, early access, Sep. 9, 2020, doi: 10.1109/TC.2020.3023045.

[2] D. J. Bernstein, "Introduction to post-quantum cryptography," in *Post-Quantum Cryptography* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2009, pp. 1–14.

[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.

[4] TNI Standards and NIST. *Post-Quantum Cryptography Standardization, 2017–2018*. Accessed: May 20, 2021. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[5] D. Jao *et al.*, "Supersingular isogeny key encapsulation," NIST Post-Quantum Standardization Project, Gaithersburg, MD, USA, Tech. Rep., 2017. [Online]. Available: https://sike.org/

[6] D. Jao and L. D. Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. Int. Workshop Post-Quantum Cryptogr.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2011, pp. 19–34.

[7] H. Fujii and D. F. Aranha, "Curve25519 for the Cortex-M4 and beyond," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2017, pp. 109–127.

[8] H. Seo, "Memory efficient implementation of modular multiplication for 32-bit ARM Cortex-M4," *Appl. Sci.*, vol. 10, no. 4, p. 1539, Feb. 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 *Proc. IEEE 27th Symp. Comput. Arithmetic (ARITH)*, Jun. 2020, pp. 72–79.

[10] M. B. Niasar, R. Azarderakhsh, and M. M. Kermani, "Efficient hardware implementations for elliptic curve cryptography over Curve448," in *Proc. Int. Conf. Cryptol. India*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2020, pp. 228–247.

[11] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the security of supersingular isogeny cryptosystems," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2016, pp. 63–91.

[12] L. D. Feo, "Mathematics of isogeny based cryptography," vol. 12, 2017, *arXiv:1711.04062*. [Online]. Available: http://arxiv.org/abs/1711.04062

[13] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki–Okamoto transformation," in *Theory of Cryptography* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2017, pp. 341–371.

[14] E. Fujisaki and T. Okamoto, "How to enhance the security of public-key encryption at minimum cost," in *Proc. Int. Workshop Public Key Cryptogr.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 1999, pp. 53–68.

[15] H. Seo, Z. Liu, P. Longa, and Z. Hu, "SIDH on ARM: Faster modular multiplications for faster post-quantum supersingular isogeny key exchange," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, no. 3, pp. 1–20, Aug. 2018.

[16] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani, "NEON-SIDH: Efficient implementation of supersingular isogeny Diffie–Hellman key exchange protocol on ARM," in *Proc. Int. Conf. Cryptol. Netw. Secur.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2016, pp. 88–103.

[17] H. Seo, P. Sanal, A. Jalali, and R. Azarderakhsh, "Optimized implementation of SIKE round 2 on 64-bit ARM Cortex-A processors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2659–2671, Aug. 2020.

[18] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Efficient and fast hardware architectures for SIKE round 2 on FPGA," Cryptol. ePrint Arch., IACR, USA, Tech. Rep. 2020/611, 2020.

[19] B. Koziel, A.-B. Ackie, R. E. Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "SIKE'd up: Fast hardware architectures for supersingular isogeny key encapsulation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 12, pp. 4842–4854, Dec. 2020.

[20] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Highly optimized Montgomery multiplier for SIKE primes on FPGA," in *Proc. IEEE 27th Symp. Comput. Arithmetic (ARITH)*, Jun. 2020, pp. 64–71.

[21] T. Lorenser, "The DSP capabilities of ARM Cortex-M4 and Cortex-M7 processors," DSP Feature Set Benchmarks, ARM, U.K., White Paper, 2016.

[22] ARM. *Cortex-M4 ISA*. Accessed: Mar. 29, 2021. [Online]. Available: https://developer.arm.com/documentation/100166/0001

[23] 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., IACR, USA, Tech. Rep. 2018/932, 2018, p. 932.

[24] SIDH Library. *PQCrypto V3.3*. Accessed: May 20, 2021. [Online]. Available: https://github.com/Microsoft/PQCrypto-SIDH

[25] H. Seo, A. Jalali, and R. Azarderakhsh, "SIKE round 2 speed record on ARM Cortex-M4," in *Proc. Int. Conf. Cryptol. Netw. Secur.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2019, pp. 39–60.

[26] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk SSSR*, vol. 145, no. 2, pp. 293–294, 1962.

[27] F. D. Santis and G. Sigl, "Towards side-channel protected X25519 on ARM Cortex-M4 processors," *Proc. Softw. Perform. Enhancement Encryption Decryption, Benchmarking*, Utrecht, The Netherlands, 2016, pp. 19–21.

[28] M. Hutter and E. Wenger, "Fast multi-precision multiplication for public-key cryptography on embedded microprocessors," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2011, pp. 459–474.

[29] H. Seo and H. Kim, "Multi-precision multiplication for public-key cryptography on embedded microprocessors," in *Proc. Int. Workshop Inf. Secur. Appl.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2012, pp. 55–67.

[30] H. Seo and H. Kim, "Consecutive operand-caching method for multiprecision multiplication, revisited," *J. Inf. Commun. Converg. Eng.*, vol. 13, no. 1, pp. 27–35, Mar. 2015.

[31] B. Haase and B. Labrique, "AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 2, pp. 1–48, Feb. 2019.

[32] E. Alkim *et al.*, "Polynomial multiplication in NTRU prime," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 1, pp. 217–238, Dec. 2020.

[33] M. Scott and P. Szczechowiak, "Optimizing multiprecision multiplication for public key cryptography," IACR Cryptol. ePrint Arch., IACR, Tech. Rep. 2007/299, 2007, p. 299.
[34] Y. Lee, I.-H. Kim, and Y. Park, "Improved multi-precision squaring for low-end RISC microcontrollers," *J. Syst. Softw.*, vol. 86, no. 1, pp. 60–71, 2013.
[35] H. Seo, Z. Liu, J. Choi, and H. Kim, "Multi-precision squaring for public-key cryptography on embedded microprocessors," in *Proc. Int. Conf. Cryptol. India*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2013, pp. 227–243.
[36] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie–Hellman," in *Proc. Annu. Int. Cryptol. Conf.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2016, pp. 572–601.
[37] Z. Liu and J. Großschädl, "New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers," in *Proc. Int. Conf. Cryptol. Afr.*, in Lecture Notes in Computer Science. Berlin, Germany: Springer, 2014, pp. 215–234.
[38] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[39] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4," IACR, USA, Tech. Rep. 2019/844, 2019.
[40] M.-J. O. Saarinen, "Mobile energy requirements of the upcoming NIST post-quantum cryptography standards," in *Proc. 8th IEEE Int. Conf. Mobile Cloud Comput., Services, Eng. (MobileCloud)*, Aug. 2020, pp. 23–30.
[41] M.-S. Chen and T. Chou, "Classic McEliece on the ARM Cortex-M4," IACR, USA, Tech. Rep. 2021/492.

Mila Anastasova graduated in computer science and engineering from the University Carlos III of Madrid, Spain, in 2019. She is currently pursuing the Ph.D. degree in computer engineering with the Institute for Sensing and Embedded Network Systems Engineering (I-SENSE), Florida Atlantic University, USA. She is also forming part of I-SENSE, Florida Atlantic University. She is researching in the area of isogeny-based quantum secure cryptography.

Reza Azarderakhsh (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Western University in 2011. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Florida Atlantic University. His current research interests include finite field and its applications, elliptic curve cryptography, pairing-based cryptography, and post-quantum cryptography. He was a recipient of the NSERC Post-Doctoral Research Fellowship working with the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. He is serving as an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS.

Mehran Mozaffari Kermani (Senior Member, IEEE) received the B.Sc. degree in electrical and computer engineering from the University of Tehran, Tehran, Iran, in 2005, and the M.E.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Western Ontario, London, Canada, in 2007 and 2011, respectively. He is currently with the Computer Science and Engineering, University of South Florida (USF), Tampa, FL, USA. His current research interests include emerging security measures for embedded systems, fault diagnosis in cryptographic hardware, and low-power secure and efficient FPGA and ASIC designs. He was a recipient of the prestigious Natural Sciences and Engineering Research Council of Canada (NSERC) Post-Doctoral Research Fellowship. From 2014 to 2015, he served as the Guest Editor for the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING Special Issue on Emerging Security Trends for Deeply-Embedded Computing Systems.