Time-Memory Trade-Offs for Saber+ on Memory-Constrained RISC-V Platform

Jipeng Zhang, Junhao Huang, Zhe Liu, *Senior Member, IEEE*, and Sujoy Sinha Roy

Abstract—Saber is a module-lattice-based key encapsulation scheme that has been selected as a finalist in the NIST Post-Quantum Cryptography standardization project. As Saber computes on considerably large matrices and vectors of polynomials, its efficient implementation on memory-constrained IoT devices is very challenging. In this paper, we present an implementation of Saber with a minor tweak to the original Saber protocol for achieving reduced memory consumption and better performance. We call this tweaked implementation 'Saber+', and the difference compared to Saber is that we use different generation methods of public matrix A and secret vector s for memory optimization. Our highly optimized software implementation of Saber+ on a memory-constrained RISC-V platform achieves 48% performance improvement compared with the best state-of-the-art memory-optimized implementation of original Saber. Specifically, we present various memory and performance optimizations for Saber+ on a memory-constrained RISC-V microcontroller, with merely 16KB of memory available. We utilize the Number Theoretic Transform (NTT) to speed up the polynomial multiplication in Saber+. For optimizing cycle counts and memory consumption during NTT, we carefully compare the efficiency of the complete and incomplete-NTTs, with platform-specific optimization. We implement 4-layers merging in the complete-NTT and 3-layers merging in the 6-layer incomplete-NTT. An improved on-the-fly generation strategy of the public matrix and secret vector in Saber+ results in low memory footprint. Furthermore, by combining different optimization strategies, various time-memory trade-offs are explored. Our software implementation for Saber+ on selected RISC-V core takes just 3,809K, 3,594K, and 3,193K clock cycles for key generation, encapsulation, and decapsulation, respectively, while consuming only 4.8KB of stack at most.

Index Terms—NTT, saber, memory optimizations, RISC-V, post-quantum cryptography, lattice-based cryptography

1 Introduction

As the NIST Post-Quantum Cryptography (PQC) standardization process is coming to an end, the deployment of PQC schemes in real-world applications has become a research hotspot in cryptographic engineering, especially in the Internet of Things (IoT) scenarios millions of IoT devices have appeared in our daily life. Since UC Berkeley initiated the RISC-V project in 2010, RISC-V has been greatly developed due to its open-source and extensible properties. With the collaboration of over 1000 international members, including many famous companies such as Western Digital, Qualcomm, Alibaba, Huawei [1], over 80 mature RISC-V chips have been manufactured, and many of them have been prevalently used in IoT scenarios. The implementation of PQC on RISC-V also attracts much attention. Many hardware/software co-design based on RISC-V for PQC schemes have been conducted [2], [3]. The NIST Lightweight Cryptography Workshop (LWC) also uses RISC-V chips to benchmark lightweight cryptography. However, the availability of PQC on IoT devices faces many challenges due to the limited resources, low power consumption, and low frequency of IoT devices. Many IoT devices have extremely limited resources, especially in Wireless Sensor Network (WSN), which consists of millions of dedicated sensors acting as environmental monitoring or target tracking [4], and the available RAM of these commercial sensors is about 4KB-32KB[5, Sec 5.4.7]. In order to solve the problem of PQC deployment on memory-constrained RISC-V chips, this paper provides a compact and optimized PQC implementation for the RISC-V chips.

Saber [6], one of the four key establishment finalists, is based on the Module Learning with Rounding (Mod-LWR) problem. Saber is quite similar to the Mod-LWE (Module Learning with Errors [7]) based scheme Kyber [8]. Both of them introduce a small-dimension matrix, contributing to a more flexible PQC scheme than ideal lattice-based [9] schemes such as NewHope [10]. Nevertheless, the public matrix in Saber and Kyber also increases the difficulty of deployment in IoT devices. The significant difference between Saber and Kyber is that Saber uses a power-of-two modulus ($q = 2^{13}$), which eliminates complex rejection sampling and modular reduction. However, this modulus precluded the usage of the Number Theoretic Transform (NTT) from accelerating polynomial multiplication in the original proposal. The original polynomial multiplication, Toom-Cook-Karatsuba-Schoolbook adopted by Saber, is asymptotically slower than NTT

- *Jipeng Zhang and Zhe Liu are with the Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 211106, China. E-mail: jp-zhang@outlook.com, zhe.liu@nuaa.edu.cn.*
- *Junhao Huang is with the Beijing Normal University-Hong Kong Baptist University United International College, Zhuhai, Guangdong 519087, China. E-mail: jhhuang_nuaa@126.com.*
- *Sujoy Sinha Roy is with the Graz University of Technology, 8010 Graz, Austria. E-mail: sujoy.sinharoy@iaik.tugraz.at.*

Manuscript received 8 Oct. 2021; revised 18 Dec. 2021; accepted 9 Jan. 2022. Date of publication 14 Jan. 2022; date of current version 10 Oct. 2022.
This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0107703, in part by the National Natural Science Foundation of China under Grant 62132008 and CCF-Tencent Open Fund. The work of Jipeng Zhang was supported in part by 2021 Tencent Rhino-Bird Research Elite Training Program.
(Corresponding author: Zhe Liu.)
Recommended for acceptance by D. Liu.
Digital Object Identifier no. 10.1109/TC.2022.3143441

and consumes more memory, which is unfriendly to memory-constrained devices. Therefore, the introduction of NTT for Saber allows us to speed up Saber's polynomial multiplication on memory-constrained devices [11].

Motivation. This paper aims to explore various time-memory trade-offs on memory-constrained RISC-V-based devices. Polynomial multiplication is not only a critical performance factor but also affects the memory footprint of Saber. The Toom-Cook-based polynomial multiplication in the Saber's reference implementation [12] and related optimization work [13], [14] achieves excellent performance, but its memory footprint is quite large. [13, Sec 2.2.3] and [14, Sec 4.2] use four levels of Karatsuba recursion to achieve very small memory consumption, but there is a significant performance penalty. NTT's adaptation for Saber [11, Sec 3.1] opens a new window for exploring various time-memory trade-offs due to significant advantages of NTT over Toom-Cook in terms of time and memory. However, Saber's original incompatibility with NTT brings some memory penalties. For example, in the Saber.PKE.KeyGen stage, we need to store the original secret vector for subsequent usage and also store the NTT-domain counterpart for computing matrix-vector multiplication. For Saber, only 32-bit NTT is feasible, so storing polynomials in the NTT domain has a higher memory footprint than the 16-bit NTT in Kyber and NewHope. The performance optimization of NTT has been thoroughly studied, including optimization on the RISC-V platform [15]. However, the performance of NTT in Saber is not the only goal. Its memory optimization is also crucial. Moreover, there is no relevant work to study how to eliminate the side effects of NTT's adaptation for Saber as far as we know.

Contributions. Our contributions in this work can be summarized as below:

1) We introduce two tweaked GenA and a tweaked GenS strategies over the framework of Saber, and we call the tweaked Saber as Saber+. More specifically, we propose an on-the-fly generation strategy for Saber+'s secret vector (on-the-fly GenS). This technique allows us to store merely one polynomial instead of the entire secret vector. We improve previous on-the-fly generation of the public matrix (on-the-fly GenA) proposed in [13, Sec 2.2.1]. Besides, inspired by Kyber's GenA implementation[1], we propose an out-of-order GenA technique for Saber+. This technique breaks the restriction on the generation order of the public matrix, and its combination with the on-the-fly GenS achieves an excellent time-memory trade-off.
2) Based on the techniques mentioned above and different memory allocation schemes for the secret vector, we propose three strategies for computing matrix-vector multiplication. These three strategies present different time-memory trade-offs, and it is worth mentioning that they can perfectly meet the individual memory requirements of LightSaber+, Saber+, and FireSaber+.
3) We review the selection of NTT parameters and construct a new set of parameters for Saber+, which can also be used in the original Saber. Our new selection can eliminate the modular reduction after the addition operation in Gentleman-Sande butterflies and achieve better performance in the 6-layer NTT implementation.
4) We carefully analyze the efficiency of the complete-NTT and various incomplete-NTTs from both arithmetic and implementation perspectives. We implement the complete-NTT, 7-layer NTT, 6-layer NTT, and 5-layer NTT with hand-written assembly and conclude that the 6-layer NTT is most efficient for all variants of Saber+ on the selected platform. All our NTT optimizations are also suitable for the original Saber.

Same with Saber's reference implementation, our implementation is constant-time and does not have any secret dependent branching or secret dependent memory accesses.

Organization of the Paper. Section 2 describes the Saber scheme, polynomial multiplication, and our target platform. Section 3 presents the efficient implementation of Montgomery reduction, the parameters selection of NTT, comparisons of the complete-NTT and various incomplete-NTTs, and efficient layers merging techniques. Section 4 describes our improved on-the-fly generation of the public matrix and secret vector, explores various time-memory trade-offs, and clarifies the differences between Saber+ and Saber. In Section 5, we compare the performance and stack usage of our optimized implementation with previous work.

Availability of Our Software. All source codes are available at https://github.com/Ji-Peng/Saber_RV32

2 Preliminaries

2.1 Saber KEM

Saber [6], [12], due to its high security, flexibility, and simplicity, was selected for the final round and became one of the four KEM finalists in the NIST PQC standardization competition. The IND-CCA secure Saber.KEM scheme is based on the IND-CPA secure public-key encryption (Saber.PKE) scheme with the help of Fujisaki-Okamoto transformations [16] and some symmetric cryptographic primitives [17]. The IND-CPA secure Saber.PKE scheme is illustrated in Algorithms 1, 2, and 3 and we refer the readers to [12] for details about Saber.KEM scheme since the optimizations in this paper mainly applied to the Saber.PKE scheme.

Saber and Kyber, as two of the four KEM finalists, are both constructed with module lattices, which demonstrates that the security and efficiency of the PQC schemes based on module lattices have been widely recognized compared with the pure LW{E,R} [18], [19] or Ring-LW{E,R} [9] problems. The IND-CPA security of Saber.PKE is reduced from the Mod-LWR problem [7], which is a module version of the LWR problem by introducing a small l-dimensional matrix and two fixed power-of-two moduli $p,\ q$. Formally, a Mod-LWR sample is given by

$$\left(\boldsymbol{A}, b = \left\lfloor \frac{p}{q}\left(\boldsymbol{A}^T \boldsymbol{s}\right) \right\rceil \right) \in R_q^{l\times l} \times R_q^{l\times 1} \tag{1}$$

1. gen_matrix routine in https://github.com/pq-crystals/kyber/blob/master/ref/indcpa.c

where the secret vector $\boldsymbol{s}$ is sampled from the centered binomial distribution $\beta_\mu(R_q^{l\times 1})$ and the public matrix $\boldsymbol{A}$ is sampled from the uniform random distribution $\mathcal{U}(R_q^{l\times l})$. The decisional Mod-LWR problem states that it is hard to distinguish whether the $(\boldsymbol{A}, b)$ are generated by the Mod-LWR distribution or the uniform random distribution $\mathcal{U}(R_q^{l\times l} \times R_q^{l\times 1})$. The two fixed power-of-two moduli p and q not only avoid the noise sampling and rejection sampling but also eliminate the explicit modular reduction when computing polynomial multiplication, which significantly simplifies the scheme. The three variants of Saber, namely LightSaber, Saber, and FireSaber, use the module dimensions l=2, 3, and 4. They all share the same underlying arithmetic operations, which further demonstrates Saber's flexibility. Saber defines three constant polynomials h, h_1, and h_2 to simplify rounding operations into simple shift operations. As described in line 5 of Algorithm 1 and line 6 of Algorithm 2, the noise in Saber. PKE scheme is deterministically generated by using simple and efficient shift operations to scale down from modulus q to modulus p. The parameters $\epsilon_p, \epsilon_q, \epsilon_T$ in Algorithms 1, 2, and 3 satisfy $p = 2^{\epsilon_p}, q = 2^{\epsilon_q}$ and $T = 2^{\epsilon_T}$ respectively. More details about these constants can be found in [12].

Algorithm 1. Saber.PKE.KeyGen() [12]

```
1: $\text{seed}_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\boldsymbol{A} = \text{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
3: $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: $\boldsymbol{s} = \beta_\mu(R_q^{l\times 1}; r)$
5: $\boldsymbol{b} = ((\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
6: 7: return $(pk := (\boldsymbol{b}, \text{seed}_{\boldsymbol{A}}), sk := (\boldsymbol{s}))$
```

Algorithm 2. Saber.PKE.Enc($pk = (\boldsymbol{b}, \text{seed}_{\boldsymbol{A}}), m \in R_2; r$) [12]

```
1: $\boldsymbol{A} = \text{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
2: if $r$ is not specified then
3: $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: end if
5: $\boldsymbol{s}' = \beta_\mu(R_q^{l\times 1}; r)$
6: $\boldsymbol{b}' = ((\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
7: $v' = \boldsymbol{b}^T(\boldsymbol{s}' \bmod p) \in R_p$
8: $c_m = ((v' + h_1 - 2^{\epsilon_p - 1}m) \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
9: return $c := (c_m, \boldsymbol{b}')$
```

The generation of the public matrix $\boldsymbol{A}$ (GenA) and secret vector $\boldsymbol{s}$ (GenS) in Algorithms 1 and 2 are implemented by the eXtend Output Function (XOF). In Saber, SHAKE-128 [20] is used to produce pseudo-random bytes, which are further used to generate the uniformly distributed public matrix and the centered binomial distributed secret vector. The SHAKE-128 adopts a sponge construction, which absorbs an initial seed into the Keccak state using keccak_absorb(). After that, the keccak_squeezeblocks() is used to generate the pseudo-random bytes, where the size of each pseudo-random block is 168 bytes. Depending on whether the polynomial generation sequence is the same in $\boldsymbol{A}$ and $\boldsymbol{A^T}$, GenA can be performed in two ways: in-order GenA and out-of-order GenA. Saber adopts the in-order GenA strategy while Kyber adopts the out-of-order GenA by attaching coordinates (i, j) into the initial seed, and the detailed difference is given in Section 4.4. Each strategy has its advantages and disadvantages. The first scheme only needs to absorb once, and the matrix generation fully utilizes each pseudo-random block without wasting any pseudo-random bytes. Hence, when generating the public matrix $\boldsymbol{A}$, fewer keccak_squeezeblocks() will be called. Although the out-of-order GenA causes a minor performance decrease compared with the in-order GenA, this strategy enables an excellent time-memory trade-off with the help of the on-the-fly GenS (see Sections 4.4 and 4.5).

Algorithm 3. Saber.PKE.Dec($\boldsymbol{s}, c = (c_m, \boldsymbol{b}')$ [12])

```
1: $v = \boldsymbol{b}'^T(\boldsymbol{s} \bmod p) \in R_p$
2: $m' = ((v - 2^{\epsilon_p - \epsilon_T}c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
3: return $m'$
```

2.2 Polynomial Multiplication

Although the choice of the power-of-two moduli p, q simplifies the Saber scheme, the original power-of-two modulus $q = 2^{13}$ excludes the use of the asymptotically faster NTT based multiplication. Therefore, the reference implementation of Saber [12] combined Toom-Cook and Karatsuba (TC/K) multiplication algorithms to accelerate the polynomial multiplication in R_q. Recently, Chung *et al.* [11, Sec 3.1] adapted NTT for Saber, NTRU [21], and LAC [22] and achieved superior performance compared with the previous TC/K multiplication implementation. NTT provides a perfect solution for Saber's implementation on memory-constrained IoT microcontrollers regarding memory footprint and efficiency. We refer the readers to [13, Sec 2.1] and [14, Sec 2] for the details about TC/K multiplication algorithm.

2.2.1 Number Theoretic Transform

The premise of using NTT in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ is that q is a prime number. When q satisfies $q \equiv 1 \bmod 2n$, there exists a $2n$-th primitive root of unity, thus the complete-NTT transformation is available. Considering the polynomial $X^{256} + 1$ in Kyber and Saber, the complete-NTT transformation means that we can factor the polynomial $X^{256} + 1$ into 256 polynomials of degree-0 modulo q. Let's denote the set of all 512-th roots of unity as $\{\zeta, \zeta^3, \zeta^5, \ldots, \zeta^{511}\}$. Then, the polynomial $X^{256} + 1$ can be decomposed as

$$X^{256} + 1 = \prod_{i=0}^{255}(X - \zeta^{2i+1}) = \prod_{i=0}^{255}\left(X - \zeta^{2\text{br}_8(i)+1}\right), \tag{2}$$

where $\text{br}_8(i)$ denotes the bit-reversal of an unsigned 8-bit integer i. After the complete-NTT transformation, the polynomial $f \in R_q$ is decomposed into 256 polynomials of degree-0, which can be written as

$$\text{NTT}(f) = \hat{f} = (\hat{f}_0, \hat{f}_1, \cdots, \hat{f}_{255})$$

with

$$\hat{f}_{2i} = \sum_{j=0}^{255} f_{2j}\zeta^{(2\text{br}_8(i)+1)j}, \hat{f}_{2i+1} = \sum_{j=0}^{255} f_{2j+1}\zeta^{(2\text{br}_8(i)+1)j}. \tag{3}$$

The product of two polynomials f, g can be performed by first transforming f, g into the NTT domain, and then performing the point-wise base multiplication for 256 polynomials of degree-0 modulo $X - \zeta^{2\mathrm{br}_8(i)+1}$. After that, the INTT (invert NTT) transformation is conducted to convert the NTT-domain result $\hat{h} = \hat{f} \circ \hat{g}$ back to the standard domain. The whole process can be described as $f \cdot g = \mathrm{INTT}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g))$.

Recent research [23] found that even though the prime q doesn't satisfy $2n|(q-1)$, the polynomial multiplication can still be accelerated by utilizing NTT. When q is a prime number and satisfies $n|(q-1)$ or $(n/2)|(q-1)$, there is only n-th or $n/2$-th primitive root instead of $2n$-th primitive root. Therefore, the polynomial cannot perform the complete-NTT transformation but can perform incomplete 7-layer or 6-layer NTT transformation for $n = 256$, eliminating the final one or two NTT layer(s). The NIST PQC Round2 submission of Kyber [24] adopted the incomplete 7-layer NTT by modifying the prime q from 7681 to 3329, which satisfies $256|(3329-1)$. Therefore, it terminates the NTT transformation in the final layer.

2.3 Target Platform: E31 RISC-V Core

Our target platform is the SiFive Freedom E310 board equipped with a 32-bit E31 RISC-V core. It is a real-world memory-constrained RISC-V device, with merely 16KB memory available. Its register file consists of 32 32-bit registers, among which 30 registers ($t_0 \sim t_6$, $s_0 \sim s_{11}$, $a_0 \sim a_7$, ra, gp, tp) are available for programming, while the other two registers (zero, sp) are reserved registers. The instruction set of this microcontroller belongs to RV32IMAC in RISC-V ISA standard, including the basic integer (I), integer multiplication and division (M), automatics (A), and 16-bit compressed instructions (C). Although it has more available registers than other CPU processors, there is no flag register to hold carry flag and overflow flag. Therefore, when using the basic integer instructions to compute big number addition or subtraction, an extra instruction **sltu** or **bltu** is required to handle carry or overflow, which results in significant overhead. In order to compute the sum of two 64-bit integers, at least four instructions are used, while other processors can complete it in just two instructions with the help of the carry flag. As for the RISC-V integer multiplication and division module, E31 is equipped with some 5-cycle multiplication instructions. A 32-bit multiplication is performed by **mul** and **mulh** instructions. Both **mul** and **mulh** cost 5 cycles. The instructions of RV32IMAC are inherently constant-time except the division instruction. It should be noted that most of the discussion on register usage targets optimization for the RISC-V Instruction Set Architecture (ISA) and can be applied to any 32-bit RISC-V device. Apart from that, the discussion on instruction latency, memory latency, instruction caches, and flash memory aims at the selected RISC-V devices. Our analysis method can be used for other (different micro-architecture) RISC-V platforms by tweaking the cycle counts of corresponding instructions.

One might question the selection of a RISC-V device with such limited memory. We want to emphasize our reasons for choosing E31 core. First, as the rapid development of the open-source RISC-V ISA, implementing cryptographic algorithms on RISC-V devices has become a research hotspot. The NIST LWC group also recommends RISC-V as one of the benchmark platforms. As far as we know, there are very limited researches conducted on the software implementation of PQC schemes, especially Saber, on RISC-V devices. As for the memory restriction, we believe choosing a real-world memory-constrained device will obtain realistic and accurate experimental results and provide a good reference for deploying PQC schemes in resource-constrained application scenarios.

3 Performance Optimization

In this section, we first present an optimized implementation of Montgomery reduction. We then discuss the choice of NTT parameters and carefully compare the performance of the complete-NTT and various incomplete-NTTs from arithmetic and implementation perspectives. [11, Sec 3.1] first adapted NTT-based polynomial multiplication for Saber, and their results showed that NTT outperforms previous Toom-Cook on Cortex-M4 and Intel platforms.

3.1 Efficient Implementation of the Montgomery Reduction

The Montgomery reduction can efficiently compute modular reduction without using division instructions, and its execution is constant-time. A signed Montgomery reduction was presented in [25, Alg 3]. As a result of using the 32-bit NTT, our Montgomery reduction takes a 64-bit signed integer as input and outputs a 32-bit result ranging from $-M$ to M, where M is the modulus in our NTT implementation. The details about the signed Montgomery reduction on RISC-V are given in Algorithm 4.

Algorithm 4. Signed Montgomery Reduction on 32-bit RISC-V, $\beta = 2^{32}$

```
Require: $a = a_h 2^{32} + a_l$, and $-\frac{\beta}{2}M \le a < \frac{\beta}{2}M$ where $M$ is the
    modulus in our NTT implementation
Ensure: $t = \beta^{-1} a \bmod M$, and $-M < t < M$
1: $mul\ a_l,\ a_l,\ M' \triangleright M' = M^{-1}$
2: $mulh\ a_l,\ a_l,\ M$
3: $sub\ t,\ a_h,\ a_l$
4: return $t$
```

3.2 Parameters Selection of NTT

To adapt NTT for Saber, we need to pick a prime M, which must be larger than the largest intermediate value produced by Saber's polynomial multiplication without modular reduction. The parameter μ is equal to 10, 8, and 6 for LightSaber, Saber, and FireSaber, respectively, which means that the coefficients of a noise polynomial range from -5/-4/-3 to 5/4/3. We convert the polynomial coefficients in $\mathbb{Z}_q$ into a signed centered representation for getting a more compact range boundary, and our NTT implementation is also performed over signed integers. Thanks to Saber's power-of-two modulus $q = 2^{13}$, it is very efficient to convert $a_i \in [0, q)$ to the centered representation, arithmetic shift left by 3 bits and then arithmetic shift right by 3 bits to get $a'_i \in [-\frac{q}{2}, \frac{q}{2}]$ with a 16-bit signed representation. The polynomial multiplication $a \cdot s$ without modular reduction produces a maximum intermediate value $5 \cdot \frac{q}{2} \cdot n$ and a minimum intermediate value $-5 \cdot \frac{q}{2} \cdot n$, where $a \in R_q$ and s is a noise polynomial. So we get the first restriction on M

$$M \geq 2 \cdot 5 \cdot \frac{q}{2} \cdot n = 10485760 \quad (4)$$

The forward NTT uses Cooley-Tukey (CT) butterflies, and the invert NTT uses Gentleman-Sande (GS) butterflies in our implementation. The input of CT butterflies is in normal order, while the output is in bit-reverse order. GS butterflies are just the opposite. Given two integers a and b as the input, the CT butterflies compute $a + \gamma \cdot b,\ a - \gamma \cdot b$ and the GS butterflies compute $a + b,\ (a - b) \cdot \gamma$. Montgomery multiplication is used when computing $\gamma \cdot b$ and $(a - b) \cdot \gamma$, and they are both range from $-M$ to M. So, the coefficients of CT butterflies increase by M after each addition, but they are doubled in GS butterflies. For example, suppose the input ranges from $-x$ to x, after continuous y-layer GS butterflies without modular reduction, the output ranges from $-x \cdot 2^y$ to $x \cdot 2^y$. If $x \cdot 2^y > 2^{31}$, then a modular reduction of the addition result is necessary for avoiding overflow from a 32-bit signed integer. In order to eliminate these additional modular reductions in GS butterflies, we get another restriction on M, $x \cdot 2^y < 2^{31}$, where x is the maximum value of INTT's input, and y is the number of layers of NTT.

In our implementation, only the 6-layer NTT is adopted. The input of the INTT is the result of the base multiplication, which ranges from $-2M$ to $2M$. We get the final range limit: $M \geq 2 \cdot 5 \cdot \frac{q}{2} \cdot n$ and $2M \cdot 2^6 < 2^{31}$, that is, $5 \cdot q \cdot n \leq M < 2^{24}$. In the end, our choice is $M = 10487809 = 512 \cdot 20484 + 1$ such that $2n \mid (M - 1)$ and the underlying field $\mathbb{Z}_q$ contains a $2n$-th primitive root of unity. Such M can implement both the complete-NTT and various incomplete-NTTs.

3.3 Arithmetic Analysis of Incomplete-NTTs

Recent research [23] shows that it is not necessary to pick an M, which satisfies $2n \mid (M - 1)$, to obtain the complete-NTT. When $n = 256 = 2^8$, the complete-NTT also refers to 8-layer NTT. The modulus of Kyber changed from 7681 to 3329 such that $256 \mid (3329 - 1)$ in NIST PQC Round 2 submission [24], which means that only 7-layer NTT is available. And related work [11], [26] also shows the effectiveness of incomplete-NTTs. To explain why incomplete-NTTs can even outperform the complete-NTT, we carefully analyze the overhead of the complete-NTT and the incomplete l-layer NTT ($l = 5, 6, 7$) from an arithmetic perspective.

In the following, we use $\mathsf{M64}$ to represent the multiplication of multiplying two 32-bit operands to obtain a 64-bit product, and $\mathsf{A64}$ represents a 64-bit long integer addition. On the selected RISC-V platform, $\mathsf{M64}$ is implemented by two multiplication instructions, **mul** and **mulh**, where **mul/mulh** computes the low-/high-limb of the 64-bit product and they both have a 5-cycles delay on the selected RISC-V platform. As described in Section 2.3, computing $\mathsf{A64}$ consumes 4 basic instructions.

We use $\mathsf{MontMul}$ to represent a Montgomery multiplication, whose computation consumes an $\mathsf{M64}$ and a Montgomery reduction (MontR). For example, $c = \mathsf{MontMul}(a, b)$ is equivalent to $c \equiv a \cdot b \bmod M$. According to Algorithm 4, we can express the overhead of a Montgomery reduction as $\mathsf{MontR} = 1\mathsf{M64} + 1\mathsf{A32}$, where $\mathsf{A32}$ represents all single-cycle instructions, including addition, subtraction, shift, and conditional set instructions. So the overhead of $\mathsf{MontMul}$ is $\mathsf{M64} + \mathsf{MontR} = 2\mathsf{M64} + 1\mathsf{A32}$. We use $< \mathsf{i}, \mathsf{j} >$ to briefly represent the overhead of $\mathsf{iM64} + \mathsf{jA32}$, whose computation consumes $10\mathsf{i} + \mathsf{j}$ cycles on the selected platform. For example, the overhead of $\mathsf{MontR}/\mathsf{MontMul}$ is $< 1, 1 > / < 2, 1 >$. The computation of one CT or GS butterfly consumes one $\mathsf{MontMul}$ and two $\mathsf{A32}$, i.e., $\mathsf{butterfly} = < 2, 3 >$. Each layer of NTT/INTT contains $\frac{n}{2}$ butterflies, so the overhead of one layer can be expressed as $\mathsf{layer} = \frac{n}{2} < 2, 3 > = < 256, 384 >$.

Base Multiplication in the Complete-NTT. The overhead of base multiplication in the complete-NTT is $n\mathsf{MontMul} = n < 2, 1 > = < 512, 256 >$.

Base Multiplication in the 7-Layer NTT. For the 7-layer NTT, the base multiplication is implemented by a polynomial multiplication of degree-1, that is, $c = c_0 + c_1 x = (a_0 + a_1 x) \cdot (b_0 + b_1 x) \bmod (X^2 - \gamma)$, where γ is a specific power of ζ. In detail, $c_0 = \mathsf{MontMul}(a_0, b_0) + \mathsf{MontMul}(\mathsf{MontMul}(a_1, b_1), \gamma)$ and $c_1 = \mathsf{MontR}(a_0 b_1 + a_1 b_0)$, where the calculation of c_1 uses lazy reduction technique. So the overhead of the base multiplication in the 7-layer NTT is

$$\frac{n}{2}(3\mathsf{MontMul} + 2\mathsf{M64} + 1\mathsf{MontR} + 1\mathsf{A64} + 1\mathsf{A32}) \\ = < 1152, 1152 >.$$

Base Multiplication in the 6-Layer NTT. The base multiplication of degree-3 in the 6-layer NTT can be expressed as $c = (a_0 + a_1 x + a_2 x^2 + a_3 x^3) \cdot (b_0 + b_1 x + b_2 x^2 + b_3 x^3) \bmod (X^4 - \gamma)$. Details about this base multiplication is described in Algorithm 5. The calculation of c_0, c_1, and c_2 have the same overhead because $\mathsf{MontMul} = \mathsf{M64} + \mathsf{MontR}$, and their overhead is

$$2\mathsf{MontMul} + 1\mathsf{MontR} + 3\mathsf{M64} + 2\mathsf{A64} + 1\mathsf{A32} = < 8, 12 >$$

The calculation of c_3 can make full use of the lazy reduction, and its overhead is

$$1\mathsf{MontR} + 4\mathsf{M64} + 3\mathsf{A64} = < 5, 13 >$$

Therefore, the overall overhead of the base multiplication in the 6-layer NTT is

$$\frac{n}{4}(3 < 8, 12 > + < 5, 13 >) = < 1856, 3136 >$$

Base Multiplication in the 5-Layer NTT. For the 5-layer NTT, the base multiplication is implemented by a polynomial multiplication of degree-7. According to Algorithm 5, the overhead of $c_0,\ c_1, \ldots$, and c_6 is equal, and their overhead is

$$2\mathsf{MontMul} + 1\mathsf{MontR} + 7\mathsf{M64} + 6\mathsf{A64} + 1\mathsf{A32} = < 12, 28 >$$

The overhead of c_7 is $1\mathsf{MontR} + 8\mathsf{M64} + 7\mathsf{A64} = < 9, 29 >$, so the overall overhead of the base multiplication in the 5-layer NTT is

$$\frac{n}{8}(7 < 12, 28 > + < 9, 29 >) = < 2976, 7200 >.$$

Complete-NTT Versus Incomplete-NTTs. The early termination of NTT can reduce the overhead of computing NTT and INTT, but the overhead of the base multiplication will

TABLE 1
Increased (Inc) and Reduced (Red) Overhead of Incomplete-NTTs Compared With the Complete-NTT

Incomplete-NTT	Inc Overhead	Inc Cycles	Red Cycles
7-layer NTT	$< 640, 896 >$	7,296	8,832
6-layer NTT	$< 1344, 2880 >$	16,320	17,664
5-layer NTT	$< 2464, 6944 >$	31,584	26,496

increase. So we need to judge whether the reduced overhead of computing NTT and INTT can outnumber the increased overhead of computing base multiplication. In Table 1, we give the increased and reduced overhead of incomplete-NTTs compared with the complete-NTT. For example, compared with the complete-NTT, the increased overhead of base multiplication in the 7-layer NTT is $< 640, 896 >=$ 7296 cycles. Moreover, the overhead of one layer in NTT is $< 256, 384 >= 2944$ cycles. One full polynomial multiplication is composed of 2 NTT, 1 INTT, and base multiplication, so terminating NTT 1-layer earlier can reduce the overhead of 3 layers, which consume $3 \times 2,944 = 8,832$ cycles. Obviously, the 7-layer NTT outperforms the complete-NTT by 1,536 cycles. Similarly, the 6-layer NTT is 1,344 cycles faster than the complete-NTT but slower than the 7-layer NTT. The 5-layer NTT is not a wise choice because the increased overhead is more than the reduced overhead. The above analysis aims to provide an insight into why incomplete-NTTs are faster than the complete-NTT from an arithmetic point of view. From an implementation perspective, our experimental results show that the 6-layer NTT is the fastest, because 3-layer merging in 6-layer NTT can use registers more efficiently than 4-layer merging in 7-layer NTT, and we will give a detailed discussion in Section 3.4. When computing a matrix-vector multiplication, the ratio of NTT/INTT to base multiplication is different, and we will re-discuss the choice of complete-/incomplete-NTTs in Section 4.

Algorithm 5. Base Multiplication in l'-layer NTT

```
Require: Degree-$(n'-1)$ polynomial $a = \sum_{i=0}^{i<n'-1} a_i x^i$ and
    $b = \sum_{i=0}^{i<n'-1} b_i x^i$ with $n' = 2^{8-l'}$
Ensure: $c = \sum_{i=0}^{i<n'-1} c_i x^i = a \cdot b \bmod (X^{n'} - \gamma)$
1: $c_j = \mathrm{MontMul}(\mathrm{MontR}(\sum_{i=j+1}^{i \le n'-1} a_i b_{n'+j-i}), \gamma) +$
    $\mathrm{MontR}(\sum_{i=0}^{i \le j} a_i b_{j-i})$ for $j \in [0, n'-1)$
2: $c_{n'-1} = MontR(\sum_{i=0}^{i \le n'-1} a_i b_{n'-1-i})$
```

3.4 NTT Optimization Techniques

The layer merging technique can improve the performance of NTT by reducing memory accesses. On a Cortex-M4 platform, there are only 16 general-purpose 32-bit registers, of which developers can only use 14. However, the 4-layer merging is still possible by packing sixteen 16-bit coefficients into eight 32-bit registers with the support of Single Instruction Multiple Data (SIMD) instructions [26, Sec 3.1]. On a RISC-V platform, 30 of the 32 32-bit registers are available to developers, so the 4-layer merging that needs to hold 16 coefficients is feasible. For the usage of registers, three registers are used for function parameters, two registers are used to store temporary values when computing the butterfly unit, and two registers are used to hold the constants M and M' in Montgomery reduction. The remaining 23 registers are used to control the loop and hold polynomial coefficients and twiddle factors.

3-Layer Merging. When computing the 6-layer incomplete-NTT, we merge six layers as 3+3. We need two registers to control the inner and outer loops, and the remaining registers are enough for holding eight polynomial coefficients and seven twiddle factors. The 6-layer INTT is merged in the same manner.

4-Layer Merging. When computing the 8-layer complete-NTT, we merge eight layers as 4+4. We only need one register to control the single loop, and 16 registers are used to hold the polynomial coefficients. So there are only six registers left to load twiddle factors, but in the first 4-layer merging, the same 15 twiddle factors are reused 16 times. We keep five of these twiddle factors in registers, and the remaining one register is used to load the other ten twiddle factors on demand. Merging as 3+3+2 is not cost-effective because 256 additional load and store operations are needed. The 5-layer and 7-layer NTT can be merged as 3+2 and 3+4, respectively. The 2-layer merging is easy to implement because fewer registers are required.

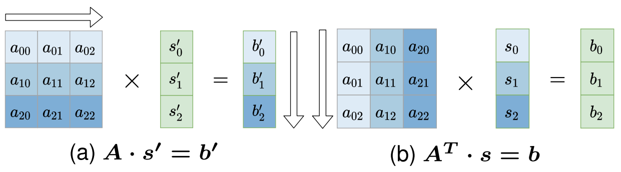
Precomputation of Twiddle Factors. It is common to store all Montgomery-domain twiddle factors in flash memory[2], but our rough estimate shows that it takes about 200 cycles to load a word from the flash memory on SiFive freedom E310. The overhead of loading all twiddle factors from flash memory is unacceptable. So we decide to store all the twiddle factors in writable memory (RAM) for better performance[3], and loading a word from the writable memory takes only two cycles on the selected platform. In this way, our link script will place and maintain the twiddle factors in a certain RAM area, occupying 0.75KB (NTT, base multiplication, and INTT need 64 32-bit twiddle factors, respectively) RAM for the 6-layer NTT. Besides, we reorder these twiddle factors in the same order as they are used. Moreover, we can reduce half of the multiplication with n^{-1} during the last layer in INTT by multiplying the last twiddle factor with n^{-1}, where $n = 64$ for the 6-layer NTT.

4 Memory Optimization

The public matrix A, secret vector s and matrix-vector multiplication consume a considerable amount of memory. Previous work [13, Sec 2.2.1] proposed an on-the-fly generation strategy for the public matrix A (on-the-fly GenA or just-in-time GenA) to reduce the memory footprint of A in Saber, and they also optimized the memory footprint during the secret vector generation. However, their on-the-fly GenA strategy is quite complicated due to the need of processing the leftover bytes. Therefore, this section improves their on-the-fly GenA by designing a simpler method to deal with the leftover bytes and proposes a new out-of-order GenA technique for Saber+. Our method does not affect the nature of the mathematical distribution of A. Moreover, we propose an on-the-fly generation strategy for the secret vector s (on-the-fly GenS). Based on the techniques mentioned above and different memory allocation schemes for the secret

2. Declaring an array with *const* keyword will suggest the linker to put it in flash memory.

3. Declaring the array without *const* keyword.



a_{00} a_{01} a_{02}
a_{10} a_{11} a_{12}
a_{20} a_{21} a_{22}
s'_0 s'_1 s'_2
b'_0 b'_1 b'_2
(a) $A \cdot s' = b'$
a_{00} a_{10} a_{20}
a_{01} a_{11} a_{21}
a_{02} a_{12} a_{22}
s_0 s_1 s_2
b_0 b_1 b_2
(b) $A^T \cdot s = b$

Fig. 1. Matrix-vector multiplication.

vector s, we present three different strategies for computing the matrix-vector multiplication in Saber+. These three strategies present different time-memory trade-offs, and they can perfectly meet the individual memory requirements of LightSaber+, Saber+, and FireSaber+.

4.1 Matrix-Vector Multiplication and On-the-Fly Generation

The matrix-vector multiplication appears twice in Saber, $A^T \cdot s$ in Algorithm 1 and $A \cdot s'$ in Algorithm 2. Taking $l = 3$ as an example, as shown in Fig. 1, $A^T \cdot s$ and $A \cdot s'$ have different computing methods. In Saber's reference implementation, the matrix A is generated in row-major order, and the matrix A^T is generated in column-major order. As shown in Fig. 1a, when computing $A \cdot s'$, the core operation is the inner product of the i-th row of the matrix A and vector s', and the result will be stored in the i-th slot of the vector b'. As shown in Fig. 1b, when computing $A^T \cdot s$, all polynomials in the i-th column of matrix A^T are multiplied by the i-th polynomial of vector s, and l generated polynomials are respectively accumulated into the l slots of vector b.

The basic idea of on-the-fly generation is that we generate a polynomial only when used and then reuse its memory space after finishing its polynomial multiplication. For example, the computing of $A \cdot s'$ contains l^2 polynomial multiplications, where each a_{ij} is used only once, so the on-the-fly strategy can be used when generating matrix A. With the on-the-fly GenA technique, the memory footprint of matrix A is reduced from $l^2 \cdot 256 \cdot 2$B to 512B. Its side effect is that we need to keep the internal state of SHAKE-128 in memory and deal with leftover bytes.

On-the-fly strategy for $A \cdot s' = b'$. When computing $A \cdot s' = b'$, we can directly convert the result of the inner product of a row of matrix A and vector s' into the ciphertext, so the memory to store vector b' can be reduced down to one polynomial. But we need to store the entire vector s', because it will be reused l times. In this manner, when computing $A \cdot s'$, we need to compute NTT $2l^2$ times. If we store 32-bit $\hat{s'}$ (the NTT-domain representative of the vector s'), then we only need to compute NTT $l^2 + l$ times, but the memory usage of $\hat{s'}$ is as high as $l \cdot 256 \cdot 4$B. The ideal way is that the matrix A is generated in column-major order, then we can merely store one NTT-domain polynomial $\hat{s'_i}$ instead of the whole vector $\hat{s'}$. The out-of-order GenA below can meet such requirements and achieves an excellent time-memory trade-off.

On-the-fly strategy for $A^T \cdot s = b$. When computing $A^T \cdot s = b$, the on-the-fly GenS technique allows us store merely a polynomial s_i instead of the vector s, but we have to store the entire vector b. In our implementation, we store the 32-bit polynomial $\hat{s}_i$ (the NTT-domain representative of s_i) instead of the original s_i. In this way, NTT is calculated only $l^2 + l$ times instead of $2l^2$.

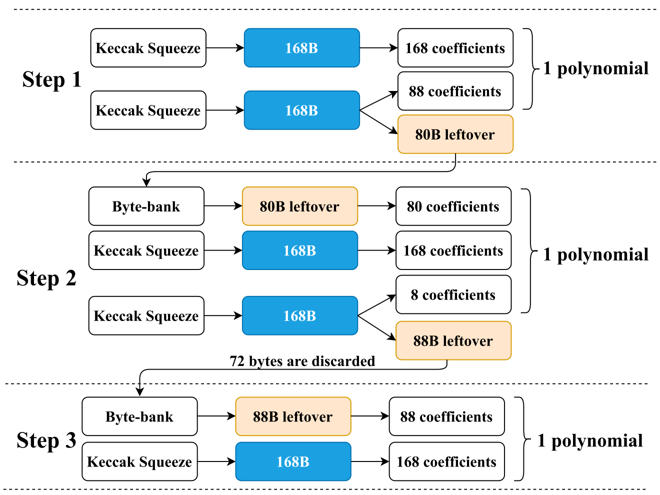


Fig. 2. On-the-fly GenS for Saber+ ($l = 3,\ \mu = 8$).

4.2 On-the-Fly GenS

Since the secret vector consists of $256 \times l$ coefficients, and one needs μ bits pseudo-random data to generate one coefficient of the secret vector using the centered binomial distribution $\beta_\mu(R_q^{l\times 1})$ where $\mu = 10, 8, 6$ for $l = 2, 3, 4$, therefore, generating a secret vector s requires 640B, 768B, and 768B pseudo-random data for LightSaber+, Saber+, and FireSaber+. We can get 168B pseudo-random data by calling the keccak_squeezeblocks() function once. So we need to call it 4, 5, and 5 times for generating a secret vector s, and there will be 32B, 72B, and 72B left for LightSaber+, Saber+, and FireSaber+. For LightSaber+, we call keccak_squeezeblocks() once and use 168B outputs to generate 128 coefficients, and the 8B leftovers are directly discarded. Discarding the 8B pseudo-random data will not affect the security of Saber+ and can simplify the generation of s. Overall, we call the keccak_squeezeblocks() 4 times to generate the secret vector s for LightSaber+. As shown in Fig. 2, we use 88B byte-bank to keep the leftover bytes for Saber+. The first 168B pseudo-random data is packed into 168 coefficients, and the second 168B is divided into two parts. The first 88B is packed into 88 coefficients and forms a complete polynomial with the 168 coefficients above, while the 80B leftovers will be kept in the byte-bank and used in Step 2. In Step 2, after using the second 168B to generate eight coefficients, there are still 160B leftovers. We discard 72B of them directly, and 72B is the maximum amount we can discard without increasing the number of keccak_squeezeblocks() calls. The remaining 88B will be used in Step 3 to generate the next polynomial. The generation strategy of FireSaber+ is similar to Saber+, except that the length of the byte-bank is 72B.

4.3 On-the-Fly GenA

The on-the-fly generation strategy of matrix A proposed in [13, Sec 2.2.1] requires a book-keeping of leftover bytes. In our implementation, we simplify this process by directly discarding some leftovers without increasing the number of calls to Keccak. Generating a polynomial coefficient requires 13-bit pseudo-random data. So, the generation of the matrix A requires 1,664B, 3,744B, and 6,656B pseudo-random data, and we need to call the keccak_squeezeblocks() 10, 23, and 40 times for LightSaber+, Saber+, and FireSaber+. We observe that every two polynomials require 5×168B pseudo-random data, i.e., each polynomial needs 2.5×168B pseudo-random data on average. Based on this observation, we propose an improved on-the-fly GenA strategy. We first pack two 168B data into 206 coefficients. After that, we use 82B of the third 168B to generate the 50 coefficients needed to form the first polynomial, and there are 86B leftovers. Then, we only keep 82B leftovers in the byte-bank for the generation of the next polynomial. In Step 2, 82B leftovers are firstly packed into 50 coefficients. Then, we only need 2×168B pseudo-random data to generate the remaining 206 coefficients of the second polynomial, and there will be no leftovers. In sum, we use 5×168B pseudo-random data to generate two polynomials. We iterate this process two and eight times to generate the matrix A for LightSaber+ and FireSaber+. For Saber+, four iterations are required to generate eight polynomials and a separate Step 1 to generate the last polynomial. Compared with the method in [13, Sec 2.2.1], our method greatly simplifies the book-keeping of leftovers without increasing the number of Keccak calls.

4.4 Out-of-Order GenA

Each entry a_{ij} in the public matrix A can be generated without order restriction in Kyber. We call their method 'out-of-order GenA'. For the generation of a_{ij}, the keccak_absorb() function takes $seed$, i, and j as input to reinitialize the internal state of Keccak, and then the outputs of keccak_squeezeblocks() are packed into a_{ij}. Hence, if we want to generate l^2 polynomials, we need to call keccak_absorb() l^2 times. In Saber's reference implementation, keccak_absorb() is called only once, and it only takes the $seed$ as input. In this way, the matrix A can only be generated in row-major order, and the matrix A^T can only be generated in column-major order. We call this method 'in-order GenA'.

In summary, the advantage of out-of-order GenA is that a_{ij} can be generated without order restriction. The disadvantage is that keccak_absorb() is called l^2 times, and the number of calls to keccak_squeezeblocks() will be increased because after generating each a_{ij}, the leftovers are directly discarded. In-order GenA has a better performance than the out-of-order GenA, but the generation order is limited.

In our implementation, out-of-order GenA is also implemented with on-the-fly strategy. When we use out-of-order GenA l^2 times to generate the matrix A, compared with the in-order GenA, the number of calls to keccak_absorb() increases by 3, 8, and 15 times, and the number of calls to keccak_squeezeblocks() increases by 2, 4, and 8 times for LightSaber+, Saber+, and FireSaber+, although the performance of out-of-order GenA is not as good as in-order GenA, its combination with the on-the-fly GenS can achieve excellent time-memory trade-offs. Based on this observation, we propose three strategies for computing the matrix-vector multiplication in Saber+.

4.5 Three Strategies for Computing Matrix-Vector Multiplication

When computing $A^T \cdot s$, on-the-fly GenA and on-the-fly GenS techniques can be used at the same time, and we only need to compute NTT $l^2 + l$ times. However, when computing $A \cdot s'$, if we use on-the-fly GenS to reduce the memory footprint of the secret vector s', then we need to compute NTT $2l^2$ times and each entry of the vector s' will be repeatedly generated l times. Based on the above analysis and proposed techniques, we propose three strategies to achieve different time-memory trade-offs. In strategy 1, we store the original vector s', so we need to compute NTT $2l^2$ times. In strategy 2, we store the NTT-domain secret vector $\hat{s'}$, and we need to compute NTT $l^2 + l$ times. In strategy 3, we use the out-of-order GenA technique to generate the matrix A in column-major order. Furthermore, with the support of on-the-fly GenS, we can merely store one NTT-domain polynomial $\hat{s}_i$, so we only need to compute NTT $l^2 + l$ times. In the above three strategies, base multiplication and INTT are computed l^2 times.

In short, strategy 1 has the smallest memory footprint but the worst performance, while strategy 2 has the best performance but the largest memory footprint. Strategy 3 achieves an excellent time-memory trade-off, and its performance and memory footprint are between the other two. In order to enable the three variants of Saber+ to be deployed on memory-constrained RISC-V-based devices, we apply strategy 1, strategy 2, and strategy 3 to FireSaber+, LightSaber+, and Saber+, respectively. LightSaber+'s memory consumption is inherently small, so strategy 2 with the largest memory footprint is suitable for LightSaber+. We apply strategy 1 with the smallest memory footprint to FireSaber+ because of its inherently large memory footprint. Saber+'s memory optimization requirements are between LightSaber+ and FireSaber+, so strategy 3 with a better time-memory trade-off is suitable. We have to note that our implementation is highly modular. The users can apply the proposed three strategies to different variants of Saber+ to get the desired time-memory trade-off.

In addition, after computing $A \cdot s'$ in Algorithm 2, s' is also used to compute the inner product $b^T \cdot s'$. In strategy 2, because we store the NTT-domain vector $\hat{s'}$, there is no need to compute $\mathrm{NTT}(s')$ again. Similarly, in strategy 3, we keep the NTT-domain polynomial $\hat{s}'_i$ until the inner product is completed. But the purpose of strategy 1 is to minimize the memory consumption, so we have to compute $\mathrm{NTT}(s')$ again in strategy 1.

In Section 3.3, we compare the efficiency of the complete-NTT and incomplete-NTTs from an arithmetic perspective. When computing the full polynomial multiplication (c=INTT(NTT(a)$\circ$NTT(b))), the ratio of NTT/INTT to base multiplication is expressed as NTT:Base=3:1. This ratio is different for matrix-vector multiplication, so we need to re-discuss the choice of incomplete-NTT. We implement all of the complete-NTT, 7-layer NTT, 6-layer NTT, and 5-layer NTT with hand-written assembly. Our experimental results show that the 6-layer NTT is the fastest for all variants of Saber+.

4.6 Saber+ Versus Saber

Note that Saber's specification doesn't explicitly specify the generation strategy of the public matrix A and the secret vector s, which means that Saber is open to different implementation strategies. Therefore, we introduce two tweaked GenA and a tweaked GenS strategies for Saber+ to obtain more time-memory trade-offs, and these tweaks on GenA and GenS are the main characteristics that distinguish Saber + from Saber. Due to different generation order of A and tweaked leftover handling strategy for A and s, our GenA and GenS strategies on Saber+, i.e., on-the-fly GenA, out-of-order GenA and on-the-fly GenS, are not compatible with the known-answer-tests (KAT) vectors of the original Saber submission. However, our tweaks do not affect the nature of the mathematical distribution, and hence they do not affect the security of Saber+. Besides, we would like to emphasize the benefits our tweaks on GenA together with our on-the-fly GenS strategy bring to Saber+. First of all, our on-the-fly GenA used in strategies 1 and 2 has a simpler mechanism to handle leftover bytes and doesn't increase the number of Keccak calls. Therefore, it has better performance in Saber+ than the Saber presented in [13, Sec 2.2.1]. Besides, the advantage of on-the-fly GenS is that it allows us to store merely one polynomial in Saber+ instead of the entire secret vector. Finally, although the out-of-order GenA used in strategy 3 causes an increase of pseudo-random bytes, its combination with on-the-fly GenS achieves an excellent time-memory trade-off for Saber+.

4.7 Others

Since the coefficients of the secret polynomials lie in a small range $[-\mu/2, \mu/2)$, where the parameter μ=10, 8, and 6 for LightSaber+, Saber+, and FireSaber+, we apply the 4-bit encoding technique proposed in [14, Sec 4.1] to the coefficients of secret polynomials.

Unlike the in-place NTT implementation in Kyber and NewHope, when computing c=INTT(NTT(a)$\circ$NTT(b)) in Saber+, apart from storing the original polynomial a and b, the corresponding 32-bit NTT-domain representations, $\hat{a}$ and $\hat{b}$, also need to be stored. When computing matrix-vector multiplication, the polynomial a is used only once, so after getting $\hat{a}$, the memory space of a can be reused to store $\hat{b}$. The memory reuse here can reduce the memory footprint of 0.5KB when computing polynomial multiplication. In order to avoid introducing an additional memory footprint, our base multiplication is implemented in place. Similar to [14, Sec 4.2], we also implement in-place verification of the decryption.

5 Results and Comparison

In order to comprehensively compare our implementation with others, we give detailed results and throughout comparison on three platforms: the selected RISC-V platform (SiFive Freedom E310 board), the simulated PQRISCV platform[4] and Cortex-M4. The first platform is a real memory-constrained platform with only 16KB of RAM, while the PQRISCV platform is a simulated RISC-V platform with 128KB of RAM.

5.1 Experimental Setup

SiFive Freedom E310. This board contains an E31 RISC-V core with RV32IMAC instruction set, but it is not equipped with a random number generator. As far as we know, there are currently no such RISC-V devices with merely 16KB memory that support a hardware random number generator. Therefore, similar to Saber's reference implementation, we use the *Counter* mode of AES (AES_CTR) to generate seed bytes.

We use SiFive GCC 8.3.0 toolchain to compile our source code with the -Os flag. The -O3 flag is the most commonly used option for better performance, but the -Os option in our implementation has better performance than the -O3 flag. The reason is that the E31 core supports an Instruction Tightly Integrated Memory (ITIM) with a maximum size of 8 KB. ITIM provides high-performance and predictable instruction delivery. Fetching an instruction from ITIM is as fast as an instruction-cache hit [27, Sec 3.1.1]. The executable program compiled with the -O3 flag cannot be totally placed in ITIM, part of it has to be placed in ROM, and fetching instructions from ROM has a longer delay. Therefore, a smaller executable program compiled with the -Os flag is more suitable for the selected RISC-V core.

PQRISCV. The PQRISCV platform targets the VexRiscv[5] implementation of the RISC-V ISA. This platform has enough memory resources, so we can run the generic C implementation in [12], [13], [14] on this platform to get a thorough comparison. We use SiFive GCC 8.3.0 toolchain to compile our source code with the -O3 flag. This platform does not provide a hardware random number generator, so we also use AES_CTR to generate seed bytes.

Cortex-M4. Our implementation is compiled and run in the same conditions as in pqm4 [28]. [11] is the known fastest implementation of Saber using NTT on the Cortex-M4 platform. In order to explore the performance penalty of our memory optimization, we ported their Cortex-M4 assembly implementation of NTT into our work.

TABLE 2
Performance Comparison for Different Generation Strategies of Public Matrix A and Secret Vector s.

Method	Cycles
GenA ([12])	923 k cycles
on-the-fly GenA (This work)	944 k cycles
on-the-fly GenA ([13])	1 296 k cycles
out-of-order GenA (This work)	1 140 k cycles
GenS ([12])	209 k cycles
on-the-fly GenS (This work)	214 k cycles

5.2 Comparison of Matrix and Secret Generation Strategies

In order to explore the performance loss of the on-the-fly generation strategies, we report their cycle counts in Table 2. The GenA and GenS are taken from Saber's reference implementation [12]. Taking $l = 3$ as an example, we call the on-the-fly GenA, out-of-order GenA nine times, and on-the-fly GenS three times. The results show that the performance loss of on-the-fly GenA and on-the-fly GenS is negligible. Although the performance penalty of out-of-order GenA is more significant than GenA or on-the-fly GenA, the increased cycles only account for about 5% of the KenGen stage. Besides, our improved on-the-fly GenA is 27% faster than [13, Sec 2.2.1] thanks to our simpler mechanism of handling leftovers. The generation of the public matrix and secret vector does not touch the memory peak, so there is no need to report their stack usage.

4. https://github.com/mupq/pqriscv

5. https://github.com/SpinalHDL/VexRiscv

TABLE 3
Comparison of Execution Time (in k Cycles) on Three Platforms

Platform	Implementation	KeyGen			Encaps			Decaps		
		$l=2$	$l=3$	$l=4$	$l=2$	$l=3$	$l=4$	$l=2$	$l=3$	$l=4$
SiFive Freedom E310	[13] (M0-mem)	-	6 546	-	-	6 614	-	-	7 133	-
	Our Saber+(C)	2 219	3 647	5 056	2 202	3 881	6 076	2 030	3 755	6 019
	Our Saber+ (Assembly)	2 560	3 809	5 063	2 205	3 594	5 360	1 773	3 193	4 928
	Speed-up (Assembly Versus Old)	—	+42%	—	—	+46%	—	—	+55%	—
PQRISCV	[12](C)	6 308	10 699	16 405	6 769	11 986	18 430	6 677	12 196	19 036
	[13] (M0-mem)	-	12 085	-	-	13 837	-	-	14 521	-
	[14](C)	5 293	8 487	11 741	5 233	8 627	12 589	4 651	8 039	12 087
	Our Saber+(C)	5 074	8 242	11 233	4 847	8 498	12 915	4 172	7 832	12 353
	Our Saber+ (Assembly)	4 494	7 000	9 083	4 020	6 885	9 587	3 012	5 720	8 360
Cortex-M4	[12](C)	1 041	2 201	3 778	1 520	2 906	4 692	1 840	3 386	5 338
	[13] (M4-mem)	-	1 165	-	-	1 530	-	-	1 635	-
	[14] (M4-mem)	607	1 233	2 052	857	1 620	2 546	973	1 765	2 749
	[11]	360	658	1 010	513	864	1 257	501	838	1 234
	Our Saber+(C)	635	1 331	2 072	888	1 693	2 913	1 018	1 887	3 184
	Our Saber+ (Assembly)	409	849	1 238	566	1 067	1 607	565	1 064	1 617

Only Saber's ($l=3$) implementation is available in [13].

TABLE 4
Comparison of Stack Usage (in Bytes) on Three Platforms

Platform	Implementation	KeyGen			Encaps			Decaps		
		$l=2$	$l=3$	$l=4$	$l=2$	$l=3$	$l=4$	$l=2$	$l=3$	$l=4$
SiFive Freedom E310	[13] (M0-mem)	-	4 984	-	-	4 696	-	-	5 800	-
	Our Saber+(C)	3 812	4 324	4 884	4 404	4 932	4 916	4 436	4 948	4 948
	Our Saber+ (Assembly)	3 804	4 316	4 876	4 396	4 924	4 908	4 428	4 940	4 940
	Reduction (Assembly Versus Old)	—	+13%	—	—	-5%	—	—	+15%	—
PQRISCV	[12](C)	9 392	13 024	19 968	11 520	15 648	23 104	12 272	16 752	24 592
	[13] (M0-mem)	-	4 980	-	-	4 068	-	-	5 172	-
	[14](C)	21 364	28 084	35 844	23 604	30 868	39 204	24 356	31 972	40 692
	Our Saber+(C)	3 796	4 416	4 852	4 372	5 008	4 900	4 420	5 024	4 932
	Our Saber+ (Assembly)	3 788	4 416	4 844	4 364	5 008	4 892	4 412	5 024	4 924
Cortex-M4	[12](C)	9 392	12 992	19 560	11 496	15 616	22 696	12 240	16 712	24 176
	[13] (M4-mem)	-	6 931	-	-	7 019	-	-	8 115	-
	[14] (M4-mem)	3 548	4 400	5 216	3 248	3 412	3 668	3 156	3 448	3 736
	[11]	14 616	23 288	37 128	16 248	32 616	40 488	16 992	33 712	41 968
	Our Saber+(C)	3 816	4 328	4 848	4 392	4 896	4 904	4 408	4 904	4 920
	Our Saber+ (Assembly)	3 800	4 312	4 832	4 376	4 880	4 888	4 392	4 888	4 904

Only Saber's ($l=3$) implementation is available in [13].

5.3 Evaluation and Comparison of Saber+ KEM

Tables 3 and 4 illustrate the experimental results of our C and assembly Saber+ KEM implementation in terms of execution time and stack usage on three platforms. In short, our memory optimization is mainly due to our tweaked GenA and GenS, which are the main difference between Saber and Saber+. Besides, combining the tweaked GenA and GenS with NTT instead of Toom-Cook to implement polynomial multiplication also reduces the memory footprint. Our performance optimization is mainly derived from NTT and its platform-specific optimizations. Note that all of our memory optimization techniques are implemented in C, and only the layer merging in NTT is implemented in assembly. As shown in Table 3, for $l=2$, the assembly-KeyGen and assembly-Encaps on SiFive E310 are slightly slower than the C implementation because the loops and functions in the assembly-NTT are expanded, resulting in a larger executable program. A smaller executable program can achieve better performance, which can be well corroborated by the fact that the software compiled with the -Os flag outperforms the -O3 flag on SiFive E310. Besides, when $l=3$ and $l=4$, the assembly-KeyGen on SiFive E310 is also slower than the C implementation. After removing AES_CTR, we find that assembly-KeyGen outperforms the C-KeyGen. We infer that due to the larger code size of the assembly-NTT, the linker puts more AES_CTR-related codes into ROM. Due to the complicated AES implementation and slower instructions fetching from ROM, a slower assembly-KeyGen and assembly-Encaps are reasonable. Compared with the C implementation, our assembly implementation shows impressive speed-up on the PQRISCV platform with relatively sufficient resources.

Since our selected RISC-V platform has merely 16KB RAM, Saber's reference implementation cannot run on SiFive E310. To highlight our optimization more clearly, we run Saber's C reference implementation [12] on the simulated PQRISCV platform. The results in Tables 3 and 4 on PQRISCV show that our C and assembly implementation is faster than the reference implementation with one-third stack usage. For $l = 3$, our assembly implementation shows $35\%, 43\%,$ and 53% speed-ups for KeyGen, Encaps, and Decaps, respectively, which clearly shows the advantage of our time-memory trade-offs. Our Saber+ implementation on Cortex-M4 also shows $40\%, 42\%,$ and 44% speed-ups compared with the reference implementation with roughly one third stack usage.

Although many related works tried to optimize Saber in terms of time and memory, only a few works focused on extremely memory-constrained scenarios (i.e., 8KB$\sim$32KB RAM available). Besides, previous memory optimizations only focused on optimizing Toom-Cook and Karatsuba [13], [14]. The NTT's adaptation on Saber was only conducted on Cortex-M4 and Intel CPU [11]. There are still no memory optimizations conducted over the NTT's adaptation on Saber. We believe our implementation can fill this gap and provide efficient time-memory trade-offs to resolve the availability of Saber in memory-constrained scenarios.

[13] from TCHES 2018 is one of the few works that explore the availability of Saber on ARM Cortex-M0 with limited memory. We deploy their generic C implementation on SiFive E310. However, they use the hardware-specific random number generator on Cortex-M0. To be fair, we modified their implementation to use AES_CTR as the seed bytes generator and benchmarked their implementation on the selected RISC-V platform. They only provide the memory-optimized implementation of Saber without LightSaber and FireSaber. As shown in Tables 3 and 4, our stack usage is slightly larger than theirs in the Encaps stage, but our implementation is 42%, 46%, 55%, and 48% faster than theirs for KenGen, Encaps, Decaps, and the entire scheme respectively. The experimental results on PQRISCV and Cortex-M4 can also draw similar conclusions.

[14] provides time-memory trade-offs for Saber on Cortex-M4, and they reported that Saber's KEM scheme could execute with less than 3.5KB stack usage thanks to their memory optimization. They provide a generic C version and a memory-efficient version (M4-mem), and the latter version optimizes the 4-level memory-efficient Karatsuba algorithm on Cortex-M4 using assembly language. In order to compare with their implementation on the RISC-V platform, we tried to deploy their generic C code on our selected RISC-V platform and PQRISCV platform. However, the stack usage of their C implementation is too large to run on SiFive E310, and their C implementation accounts for nearly 21KB-40KB of stack on PQRISCV. The performance results on PQRISCV show that our assembly implementation outperforms than theirs by $18\%, 20\%,$ and 29% for KenGen, Encaps, and Decaps, respectively, while consuming nearly one-seventh stack. For our generic C implementation, our encapsulation and decapsulation are 3% slower than the work in [14] when $l = 4$, but our stack usage is only 12%-14% of theirs. Compared with their M4-mem implementation on Cortex-M4, our implementation is 1.48-1.72 times faster than theirs. For stack usage, our KeyGen is better than theirs only when $l = 3$ or $l = 4$, and in other cases, our stack usage is 252-1468 bytes more than theirs. In summary, we believe it is worth trading these stack consumption for such a significant performance gain.

The implementation in [11, Sec 3.1] is the first work that utilizes NTT to accelerate Saber. Their results show that Saber's NTT adaptation can achieve a great performance gain. However, they did not conduct any memory optimization, which makes their implementation impractical in memory-constrained scenarios. According to Table 4, their implementation requires 14KB-41KB of stack. After integrating our memory optimizations with their NTT assembly implementation on Cortex-M4, we can save 73%-88% stack with merely 10%-31% performance loss. This firmly confirms the effectiveness of our memory optimizations.

6 Conclusion

In this paper, we presented the first known tailored RISC-V implementation of a modified Saber (Saber+). Our implementation aimed at memory optimization, and we explored various time-memory trade-offs on a memory-constrained RISC-V platform. Our time-memory trade-offs can reduce the stack consumption by 73%-88% with merely 10%-31% performance loss compared with the fastest Saber NTT adaptation implementation. Compared with previous memory-optimized implementation using 4-level memory-efficient Karatsuba, the stack consumption is at the same level, but our speed is much faster than them. Our work shows that the combination of NTT and various memory optimization techniques proposed in this paper can effectively reduce Saber+'s stack consumption, making it possible to deploy all variants of Saber+ on memory-constrained devices.

References

[1] RISC-V, "RISC-V international," 2019. [Online]. Available: https://riscv.org/

[2] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2020, pp. 239–280, 2020.

[3] G. Xin *et al.*, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2672–2684, Aug. 2020.

[4] J. Yick, B. Mukherjee, andD. Ghosal, "Wireless sensor network survey," *Comput. Netw.*, vol. 52, no. 12, pp. 2292–2330, 2008.

[5] S. K. Sharma, B. Bhushan, R. Kumar, A. Khamparia, and N. C. Debnath, *Integration of WSNs Into Internet of Things: A Security Perspective*. Boca Raton, FL, USA: CRC Press, 2021.

[6] J. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *Progress in Cryptology - AFRICACRYPT 2018*. Berlin, Germany: Springer, 2018, pp. 282–305.

[7] A. LangloisandD. Stehlé, "Worst-case to average-case reductions for module lattices," *Des. Codes Cryptogr.*, vol. 75, no. 3, pp. 565–599, 2015.

[8] J. W. Bos, *et al.*, "CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2018, pp. 353–367.

[9] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology - EUROCRYPT 2010*, H. Gilbert, Ed., Berlin, Germany: Springer, 2010, pp. 1–23.

[10] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A New Hope," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 327–343.

[11] C. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C. Shih, andB. Yang, “NTT multiplication for NTT-unfriendly rings new speed records for Saber and NTRU on Cortex-M4 and AVX2,” *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2021, no. 2, pp. 159–188, 2021.
[12] A. Basso *et al.*, “SABER: Mod-LWR based KEM (Round 3 Submission),” 2020. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions
[13] A. Karmakar, J. M. B. Mera, S. S. Roy, andI. Verbauwhede, “Saber on ARM CCA-secure module lattice-based key encapsulation on ARM,” *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2018, no. 3, pp. 243–266, 2018.
[14] J. M. B. Mera, A. Karmakar, andI. Verbauwhede, “Time-memory trade-off in Toom-Cook multiplication: An application to module-lattice based cryptography,” *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2020, no. 2, pp. 222–244, 2020.
[15] D. O. C. Greconici, “Kyber on RISC-V,” M.S. thesis, Dept. Inst. Comput. Inf. Sci. Digit. Secur., Radboud Univ. Nijmegen, Nijmegen, The Netherlands, 2020.
[16] D. Hofheinz, K. Hövelmanns, and E. Kiltz, “A modular analysis of the Fujisaki-Okamoto transformation,” in *Proc. Theory Cryptogr. Conf.*, 2017, pp. 341–371.
[17] H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma, “Post-quantum IND-CCA-secure KEM without additional hash,” *IACR Cryptol. ePrint Arch.*, vol. 2017, 2017, Art. no. 1096.
[18] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *Proc. 37th Annu. ACM Symp. Theory Comput.*, 2005, pp. 84–93.
[19] A. Banerjee, C. Peikert, and A. Rosen, “Pseudorandom functions and lattices,” in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2012, pp. 719–737.
[20] M. J. Dworkin, “SHA-3 standard: Permutation-based hash and extendable-output functions,” 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.202
[21] C. Chen *et al.*, “NTRU - Algorithm specifications and supporting documentation (Round 3 Submission),” Tech. Rep. 2020. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions
[22] X. Lu *et al.*, “LAC: Practical Ring-LWE based public-key encryption with byte-level modulus,” *IACR Cryptol. ePrint Arch.*, vol. 2018, 2018, Art. no. 1009.
[23] V. LyubashevskyandG. Seiler, “NTTRU: Truly fast NTRU using NTT,” *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2019, no. 3, pp. 180–201, 2019.
[24] R. Avanzi *et al.*, “CRYSTALS-Kyber - algorithm specifications and supporting documentation (round 2 submission),” 2019. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions
[25] G. Seiler, “Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography,” *IACR Cryptol. ePrint Arch.*, vol. 2018, 2018, Art. no. 39.
[26] E. Alkim, Y. A. Bilgin, M. Cenk, andF. Gérard, “Cortex-M4 optimizations for {R, M} LWE schemes,” *IACR Trans. Cryptogr. Hardware Embedded Syst.*, vol. 2020, no. 3, pp. 336–357, 2020.
[27] SiFive, Sifive FE310-G002 Manual, 2021. [Online]. Available: https://starfivetech.com/uploads/fe310-g002-manual-v1p0.pdf.
[28] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, PQM4: Post-quantum crypto library for the ARM Cortex-M4, 2018. [Online]. Available: https://github.com/mupq/pqm4

Jipeng Zhang received the bachelor’s degree from the Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China, in 2020. He is currently working toward the PhD degree with NUAA, Nanjing, China.

Junhao Huang received the bachelor’s and master’s degrees from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2018, and 2021, respectively. He is currently working toward the PhD degree with Beijing Normal University-Hong Kong Baptist University United International College, Zhuhai, China.

Zhe Liu (Senior Member, IEEE) received the BS and MS degrees from Shandong University, Jinan, China, in 2008 and 2011, respectively, and the PhD degree from the University of Luxembourg, Esch-sur-Alzette, Luxembourg, in 2015. He is currently a professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. His research interests include security, privacy, and cryptography solutions for the Internet of Things. He was a recipient of the prestigious FNR awards-Outstanding PhD Thesis Award in 2016, ACM CHINA SIGSAC Rising Star Award in 2017 as well as DAMO Academy Young Fellow in 2019. He served as general co-chair of CHES 2020 and CHES 2021.

Sujoy Sinha Roy received the PhD degree with ‘Summa cum laude with congratulations from the examination committee’ from COSIC, KU Leuven, Belgium, in 2017 . His doctoral thesis was awarded the ’IBM Innovation Award 2018’ that recognizes an outstanding doctoral thesis in informatics. He is currently an assistant professor with IAIK, the Graz University of Technology. He is a co-designer of ‘Saber’ which is a finalist in NIST’s Post-Quantum Cryptography Standardization Project.