Saber on ESP32

Bin Wang, Xiaozhuo Gu*, and Yingshan Yang

SKLOIS, Institute of Information Engineering, CAS, Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
wangbin171@mails.ucas.edu.cn, {guxiaozhuo, yangyingshan}@iie.ac.cn

Abstract. Saber, a CCA-secure lattice-based post-quantum key encapsulation scheme, is one of the second round candidate algorithms in the post-quantum cryptography standardization process of the US National Institute of Standards and Technology (NIST) in 2019. In this work, we provide an efficient implementation of Saber on ESP32, an embedded microcontroller designed for IoT environment with WiFi and Bluetooth support. RSA coprocessor was used to speed up the polynomial multiplications for Kyber variant in a CHES 2019 paper. We propose an improved implementation utilizing the big integer coprocessor for the polynomial multiplications in Saber, which contains significant lower software overhead and takes a better advantage of the big integer coprocessor on ESP32. By using the fast implementation of polynomial multiplications, our single-core version implementation of Saber takes 1639K, 2123K, 2193K clock cycles on ESP32 for key generation, encapsulation and decapsulation respectively. Benefiting from the dual core feature on ESP32, we speed up the implementation of Saber by rearranging the computing steps and assigning proper tasks to two cores executing in parallel. Our dual-core version implementation takes 1176K, 1625K, 1514K clock cycles for key generation, encapsulation and decapsulation respectively.

Keywords: Post-quantum cryptography · Efficient implementation · Saber · ESP32.

1 Introduction

Post-quantum cryptography has been widely developed in recent years since the public key cryptographic primitives based on traditional hard problems such as factoring or discrete logarithms are under the threat of quantum computers [28] [25]. With the goal of accelerating the research and standardization of post-quantum cryptography algorithms, the US National Institute of Standards and Technology (NIST) has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms [1].

In the NIST standardization process [2], a large number of encryption schemes against quantum attacks have been proposed, most of which are based on

* Corresponding author: guxiaozhuo@iie.ac.cn. This work has been supported by National Natural Science Foundation of China (Grant No.61602475, No.61802395) and by National Cryptographic Foundation of China (Grant No. MMJJ20170212).

the hard problems over lattices. Frodo [13], NewHope [11], Kyber [14] and Saber [16] have entered the second round of candidate processes of NIST.

ESP32, belongs to ESP series, is an embedded microcontroller, which supports WiFi and Bluetooth. And it is widely used in IoT devices. As of 2017, the shipments of ESP have reached 100-Million [4]. The report from TSR in 2018 showed ESP became a leader in the MCU Embedded WiFi chip market sector [3]. In this work, we choose ESP32 chip to implement the chosen ciphertext attack (CCA) resistant lattice-based key encapsulation mechanism (KEM) Saber [16] which has entered the second round of NIST's standardization process [9]. We make full use of the advantages of the big integer coprocessor and dual core features on ESP32 to provide efficient implementation of Saber. Our implementation mainly focuses on high performance. To the best of our knowledge, this is the first published optimized implementation of post-quantum KEM on ESP32. The source code in this paper is available in `https://github.com/SABERONESP32/SABERONESP32`.

Contribution

1 Polynomial multiplication is a very time-consuming operation in Saber. The parameters of Saber with modulus 8192 and 1024 prevent the use of the most efficient polynomial multiplication method Number Theoretic Transform (NTT) with $O(n\log n)$ complexity. RSA coprocessor was used to speed up the polynomial multiplication for Kyber [14] variant in [10]. In this work, we exploit the ability of the big integer coprocessor on ESP32 to speed up the polynomial multiplication based on the Kronecker substitution [18] which was used in [10]. We adapt the Kronecker substitution to the feature of our coprocessor and provide efficient implementations for polynomial multiplications with 13-bit and 10-bit coefficients respectively in Saber, which contains a significant lower software overhead and takes a better advantage of the big integer coprocessor on ESP32. We lower the polynomial degree by using the Karatsuba [21] and Toom-Cook [12] algorithms before Kronecker substitution to overcome the limitation of the supported bit length of our big integer coprocessor. Also we rearrange the steps of Karatsuba and Toom-Cook algorithm and assign software computing operations to CPU during its idle time using several reasonable strategies. Our fast implementation for 256 degree polynomial multiplication takes 97K, 85K clock cycles for moduli 8192 and 1024 respectively, and competes with the NTT-based polynomial multiplications which takes 244K clock cycles for the same degree with modulus 7681 on our platform. Based on the efficient implementation of polynomial multiplication, our optimized implementation of Saber takes 1639K, 2123K, 2193K clock cycles on ESP32 for key generation, encapsulation and decapsulation respectively.

2 ESP32 is an embedded microcontroller with two Harvard Architecture Xtensa LX6 CPUs. The built-in FreeRTOS [8] on ESP32 is designed for multiple tasks parallel execution and can be used to exploit the good performance provided by two cores. As ESP32 is not able to execute single task on two

cores based on hardware level instructions rearrangement, we hand-partition the CCA secure functions into small steps and rearrange them. We start the execution in main core and assign proper computing task to another core based on the computing tasks assigning api provided by FreeRTOS. Moreover, since the computing inputs of a step on one core may depend on the results of step execution on another core, we use the semaphore mechanism provided by FreeRTOS to make two cores execute steps of Saber algorithm in our expected order. As a result of executing Saber on two cores in parallel, we can reduce the 462K, 498K, 679K clock cycles for key generation, encapsulation and decapsulation respectively.

Organization. In Section 2, we briefly describe the Saber KEM scheme, the algorithms for polynomial multiplication, and our target platform. In Section 3, we introduce the Kronecker substitution algorithm and provide the implementation for it utilizing big integer coprocessor. Next, we provide our optimized implementations for polynomial multiplications and Saber in detail. Performance of our implementations and the comparisons are provided in Section 5. And the conclusion is described in final section.

2 Background

2.1 Notation

Let $\mathbb{Z}_q$ be the ring of integers with a modulus of q. $R_q = \mathbb{Z}_q[x]/(x^n+1)$ denotes the ring of integer polynomials modulo (x^n+1) , where n is a power of 2 and each coefficient of the ring is in $[0,\ q)$. The ring of $m \times n$-matrices over R is referred as $R^{m\times n}$.

The floor function $\lfloor x \rfloor$ represents the largest integer that is not greater than x, and the ceiling function $\lceil x \rceil$ represents the smallest integer that is not less than x. Moreover, $\lfloor x \rceil$ represents the nearest integer to x.

We use upper case letters to represent big integer (or "large number"), use bold lower case letters to represent vectors, and use bold upper case letters to represent matrices. For a polynomial f, we write f_i for the ith coefficient of x^i.

2.2 Saber

In this section, we briefly introduce Saber, a lattice-based post-quantum key encapsulation scheme that has entered the second round of post-quantum cryptography standardization process of the NIST [9]. Its security is based on the Module-Learning-with-Rounding (Module-LWR) problem and contains an IND-CPA encryption scheme and an IND-CCA secure key encapsulation mechanism by applying a post-quantum variant of the Fujisaki-Okamoto transform [19].

Parameters. The standard version of Saber KEM which achieves around 180-bit of quantum-security uses matrix or vector dimension $l = 3$ and ring-dimension $n = 256$. The two moduli p and q of multiplications are 2^{10} and 2^{13} respectively. The binomial error distribution uses the parameter of $\mu = 8$.

CPA secure Saber KEM. The Algorithm 5, Algorithm 6 and Algorithm 7 in the appendix demonstrate the CPA secure key generation, encryption and decryption algorithms used in Saber, respectively. The `KeyGen` function expands the random seed into the pseudorandom matrix $\boldsymbol{A}$ and is instantiated by using the extendable output function `SHAKE-128`.

CCA secure Saber KEM. The Algorithm 8 and Algorithm 9 in the appendix demonstrate the encapsulation and decapsulation operations used in the Saber KEM, respectively. The hash functions $\mathcal{G}$ and $\mathcal{H}$ are implemented using `SHA3-512` and `SHA3-256` respectively. In the following, we write `Gen`, `Enc` and `Dec` for CCA secure key generation, encapsulation and decapsulation respectively.

The performance of implementing Saber depends highly on the speed of the polynomial multiplication and the generation of $\boldsymbol{A}$ and $\boldsymbol{s}$. Here, the degree of the polynomials in Saber is 256 and the polynomial multiplication has two forms. One is matrix-vector multiplication. The matrix is composed of 3 by 3 polynomials, each having 256 13-bit coefficients. Another is vector-vector multiplication, where the coefficients are 10 bits.

2.3 Polynomial multiplication

Polynomial multiplication is a very time-consuming operation. In many implementations of post-quantum cryptographic schemes involving polynomial multiplication, number theoretic transform (NTT) is used for acceleration. However, NTT has certain limitations on the modulus. In Saber, the chosen modulus is not a prime number, so NTT-based polynomial multiplication cannot be used. The following multiplication algorithms are used in our implementation.

Karatsuba. In 1960, Karatsuba [21] proposed a fast multiplication algorithm, namely Karatsuba algorithm, which can achieve $O(n^{lg3})$ time complexity. It consists of three main phases: splitting, evaluation and interpolation. In the splitting phase, it splits the input polynomials $A(x)$ and $B(x)$ into $A(y) = A_1 \cdot y + A_0$ and $B(y) = B_1 \cdot y + B_0$, where $y = x^{n/2}$. In the evaluation phase, it evaluates w_1 to w_3 by multiplying these polynomials at the points $y = \{\infty, 1, 0\}$ respectively. In the interpolation phase, recombine these polynomials to get the final result. Through the divide-and-conquer approach, the algorithm is called recursively to get the final result. The detailed algorithm steps are shown in Algorithm 11 in the appendix.

Toom-Cook. The Toom-Cook algorithm is a generalization of Karatsuba algorithm. The implementation described by Knuth achieves the time complexity $O(n \cdot 2^{\sqrt{2lgn}} \cdot lgn)$ [23]. It splits each polynomial into w parts, each of which has n/w coefficients, so it is called w-way Toom-Cook multiplication. Following the implementations in [16] and [22], we mainly use four-way Toom-Cook multiplication shown in Algorithm 12 in the appendix, referred to as Toom-Cook4. It also contains three phases: splitting, evaluation and interpolation. The process is similar to Karatsuba. For Toom-Cook4, it splits the polynomial $A(x)$ into four parts, i.e. $A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{n/4}$, and computes its results at points $y = \{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$ as recommended in [12]. At last, combining these intermediate results yields the final polynomials.

2.4 Platform

ESP32 is an embedded microcontroller belongs to ESP Series by espressif [6]. Low price, low power consumption and built-in WiFi features (and Bluetooth on some chips) of this series of chips make it is widely used in commercial smart home products. The shipments of ESP have reached 100-Million[4] as of 2017. And the report from TSR in 2018 showed ESP became a leader in the MCU Embedded WiFi chip market sector [3]. ESP32 is based on two Harvard Architecture Xtensa LX6 CPUs running at 240MHz. There are 448 KB internal ROM and 520 KB internal SRAM on the chip. Also ESP32 has built-in both WiFi and Bluetooth support. There are development kits for audio recognition, face recognition, and applications that support Apple HomeKit provided for ESP32. In addition, ESP32 is equipped with several encryption coprocessors such as a True Random Number Generator (TRNG), a big integer coprocessor (for RSA and ECC acceleration), a SHA-2 coprocessor and an AES coprocessor. ESP32 also includes security features of secure boot and flash encryption.

As an embedded microcontroller designed for IoT environment, these coprocessors and security features built in ESP32 make it a good choice to implement post-quantum cryptographic schemes which will give it a wider prospect for security applications. In this work, we provide an efficient implementation of Saber which is a lattice-based post-quantum KEM on this device. And our main target is high speed. We consider reusing the big integer coprocessor (for RSA and ECC) to accelerate the polynomial multiplications in Saber, and scheduling the coprocessor and two CPU cores running in parallel to achieve a better performance.

3 Kronecker substitution

3.1 KS1 and KS2

The Kronecker substitution is an algorithm for computing the product of two polynomials. Since the univariate polynomial and integer arithmetic are almost identical, the Kronecker substitution converts the polynomial arithmetic to the big integer arithmetic by packing a coefficient into an integer. For example, when multiplying two polynomials $f(x) = 2x + 1$ with $g(x) = 3x + 2$ in $\mathbb{Z}[x]$, we compute the polynomials at point $x = 100$, i.e. $f(100) = 200 + 1 = 201$ and $g(100) = 300 + 2 = 302$, then multiply $201 \cdot 302 = 60702$. Corresponding to the polynomial coefficients, we can get the final polynomial multiplication result as $6x^2 + 7x + 2$. The process of converting a polynomial to an integer is called "packing". Conversely, the process of converting an integer to a polynomial is called "unpacking".

We call the standard Kronecker substitution as KS1. For two polynomials f and g of degrees m and n respectively, where $0 \leq f_i, g_i < 2^c$ for all i, the bit length of the big integers converted by KS1 needs to be $b = 2c + \lceil lg(min(m, n)) \rceil$.

In [18], David Harvey presented negated Kronecker substitution algorithm called KS2 which halves the bit length of the big integer at the cost of increasing the number of multiplications. For two polynomials f and g of degrees m and n, respectively, where $0 \leq f_i, g_i < 2^c$ for all i, the bit length of the big integers converted by KS2 needs to be $b = c + \frac{1}{2}\lceil lg(min(m, n)) \rceil$.

Different from KS1, KS2 needs to select two negated evaluation points $(2^b, -2^b)$ for multiplication, that is, perform two integer multiplications to obtain two results. The results are added to obtain the even coefficients of the final polynomial, and the results are subtracted to obtain odd coefficients.

KS1 and KS2 contain two versions of unsigned version as described above and signed version. The signed version is used for polynomial multiplications with signed coefficients. In Saber we need to perform 256 degree polynomial multiplication in $\mathbb{Z}_q$, hence in this work, we only focus on the unsigned version of Kronecker substitution for performing two polynomials with the same degree.

For KS3 and KS4 which were also presented in [18], implementing these two algorithms can further shorten the bit length of multiplications required at the cost of more complicated packing and unpacking operations. However, based on our implementation results of KS2 algorithm, the packing and unpacking routines already take time comparable to the multiplications by hardware big integer coprocessor. Hence we did not consider using KS3 and KS4 algorithms in this work.

We regard a polynomial $f \in \mathbf{Z}[x]$ with degree n, and $f = \sum_{i=0}^{n-1} f_i x^i$. We define `KSPACK` shown in Algorithm 1 and `KSUNPACK` shown in Algorithm 2 for packing and unpacking operations in Kronecker substitution.

We define `KS1MUL` shown in Algorithm 3 and `KS2MUL` shown in Algorithm 4 to compute the product of two polynomials with the same degree using KS1 and KS2 algorithm respectively. In the following description, we write `KS1MUL`(n,b) and `KS2MUL`(n,b) simply for `KS1MUL`(f,g,n,b) and `KS2MUL`(f,g,n,b) respectively, since the n and b are the primary parameters. And we write `KSMUL` as a general name for `KS1MUL` and `KS2MUL` when we does not specify which algorithm to use.

3.2 Utilizing the big integer coprocessor

On ESP32, there is a big integer coprocessor with capabilities of multiplication, modular multiplication and modular exponentiation. The coprocessor contains

Algorithm 1: KSPACK$(f, n, b, sign)$

```
Input: polynomials $f \in \mathbf{Z}[x]$
Input: degree $n$ of $f$
Input: bit length $b$ of evaluate point
Input: sign $sign \in \{+1, -1\}$ of evaluate point
Output: big integer $X$
$X \leftarrow 0$
for $i = 0, 1, \ldots, n-1$ do
    $X \leftarrow X + f_i \times (sign \times 2^b)^i$
return $X$;
```

Algorithm 2: KSUNPACK(X, n, b)

```
Input: big integer $X$
Input: degree $n$ of output polynomial
Input: bit length $b$ of output polynomial coefficients
Output: polynomial $f$
for $i = 0, 1, \ldots, n-1$ do
    $f_i \leftarrow X$ mod $2^b$
    $X \leftarrow X \ / \ 2^b$
return $f$
```

Algorithm 3: KS1MUL(f, g, n, b)

```
Input: polynomials $f, g \in \mathbf{Z}[x]$with same degree
Input: degree $n$ of $f, g$ and degree$(f)$ = degree$(g)$= $n$
Input: bit length $b$ of evaluate point (bit length of packing one coefficient)
Output: the product $h = fg$
$X \leftarrow$ KSPACK$(f, n, b, +1)$
$Y \leftarrow$ KSPACK$(g, n, b, +1)$
$Z \leftarrow X \times Y$                                // big integer multiplication
$h \leftarrow$ KSUNPCK(Z,2n-1,b)
return $h$
```

three sets of 128 registers in 32 bits for storing two inputs and one output and supports fixed bit length operations. For modular multiplication and modular exponentiation, it supports operand bit length of $N \in \{$ 512; 1024; 1536; 2048; 2560; 3072; 3584; 4096 $\}$; and for multiplication, the supported bit length is $N \in \{$ 512; 1024; 1536; 2048 $\}$ since the bit length of output is twice of the inputs.

The bit length of packing is crucial for implementing polynomial multiplication using KS1MUL and KS2MUL. For two polynomials f and g of the same degree n, with $f_i, g_i \in [0, 2^c)$, the minimum bit length b to pack one coefficient is $2c + \lceil lg(n) \rceil$ for KS1MUL, and $c + \lceil \frac{1}{2} \times lg(n) \rceil$ for KS2MUL. It is suitable for our coprocessor to compute 64 degree 13-bit coefficients polynomial multiplication using the KS1MUL algorithm (where $c = 13, b = 2c + lg(64) = 32$ and $32 * 64 = 2048$).

```
Algorithm 4: KS2MUL(f,g,n,b)
  Input: polynomials $f, g \in \mathbf{Z}[x]$with same degree
  Input: degree $n$ of $f, g$ and degree($f$) = degree($g$)= $n$
  Input: bit length $b$ of evaluate point (bit length of packing one coefficient)
  Output: the product $h = fg$
  $X_+ \leftarrow$ KSPACK($f, n, b, +1$)
  $X_- \leftarrow$ KSPACK($f, n, b, -1$)
  $Y_+ \leftarrow$ KSPACK($g, n, b, +1$)
  $Y_- \leftarrow$ KSPACK($g, n, b, -1$)
  $Z_+ \leftarrow X_+ \times Y_+$                    // big integer multiplication
  $Z_- \leftarrow X_- \times Y_-$                    // big integer multiplication
  $Z_0 \leftarrow \frac{1}{2} \times (Z_+ + Z_-)$
  $Z_1 \leftarrow \frac{1}{2} \times \frac{1}{2^b} (Z_+ - Z_-)$
  $h_0 \leftarrow$ KSUNPACK($Z_0, \lceil \frac{2n-1}{2} \rceil, 2b$)
  $h_1 \leftarrow$ KSUNPACK($Z_1, \lfloor \frac{2n-1}{2} \rfloor, 2b$)
  $h \leftarrow h_0(x^2) + x h_1(x^2)$
  return $h$
```

We pack one 13-bit coefficient into 32 bits and pack 64-degree polynomial into 64*32=2048 bits big integer and utilize the big integer coprocessor for computation. Also, the packing and unpacking are efficient since the registers of the coprocessor is exactly of 32 bits where no shifting operations required. For KS2MUL, since the KS2 algorithm can halve the number of bits that need to be packed, two times of 2048 bits big integer multiplication can be used to compute 64-degree 29-bit coefficients polynomial multiplication (where $c = 29, b = c + \frac{1}{2} lg(64) = 32$ and $32 * 64 = 2048$) and 1536 bits to compute 64 degree 21-bit coefficients polynomial multiplication (where $c = 21, b = c + \frac{1}{2} lg(64) = 24$ and $24 * 64 = 1536$).

4 Implementation

4.1 Polynomial multiplication using Kronecker substitution

In Saber, we need to compute 256-degree polynomial multiplications with 13-bit and 10-bit coefficients. We consider utilizing the big integer coprocessor based on Kronecker substitution for speeding up these operations.

The straightforward idea is as follows. We pack the two entire polynomials into big integers based on Kronecker substitution and then multiply the two big integers utilizing the coprocessor. Taking a 256 degree 13-bit coefficients polynomial multiplication as an example, for KS1MUL algorithm, one coefficient is required to be packed into $13 * 2 + lg256 = 34$ bits and the 256-degree polynomial into a big integer of $34 * 256 = 8704$ bits. The bit length is too large for direct computing by our coprocessor. The Karatsuba is an algorithm can be used for both polynomial multiplication and number multiplication with an easy implementation. We use the Karatsuba to split the big integers into small bit length which our coprocessor is able to compute. After recursive call 3 times, the bit length is 8704/2/2/2=1088. Considering the coprocessor is capable of multiplication of {512; 1024; 1536; 2048} bits, here 1536 is suitable. To reduce the overhead caused by a large amount of unaligned shifting operations for packing 34 bits, we consider packing the coefficients into 40 bits (byte aligned). And multiplying the 1536 bits is still sufficient as 40*256/2/2/2=1280. We use the mbedtls library [5] which is built in ESP32 software development kit (SDK) to perform big integer addition and shifting operations by CPU, and perform big integer multiplication by coprocessor. As a result, the software-based big integer addition and shifting operations are much less efficient than the hardware-based multiplication, and the entire process requires a total of 1180K clock cycles.

Table 1: Performance of KS1MUL and KS2MUL

Implementation[a]	Degree	Coefficient(bits)	Packing(bits)	Cycles
KS1MUL[b]	64	13	32	10,310
KS2MUL[c]	64	16	32	30,555

[a] The source code is available in our github link.

[b] Packed one 13-bit coefficient into 32 bits and required one 2048 bits big integer multiplication by the coprocessor.

[c] Packed one 16-bit coefficient into 24 bits and required two 1536 bits big integer multiplications by the coprocessor.

In the following, we consider first splitting the polynomial into low degree and then converting the low degree polynomial multiplication into big integer multiplication based on Kronecker substitution. This leads to some cheap coefficient-level operations as a trade-off for the complex software-based big integer addition and shifting operations. We discuss the polynomial multiplication with 13-bit and 10-bit coefficients respectively as follows.

256-degree 13-bit coefficients polynomial multiplication. It is suitable for our coprocessor to compute 64 degree 13-bit coefficients polynomial multiplication using the KS1MUL as described in Section 3.2, where $c = 13, b = 2c + lg(64) = 32$ and $32 * 64 = 2048$. For splitting 256-degree polynomial multiplication into 64 degree polynomial multiplication, we can use the Karatsuba algorithm of 2 recursive calls or the Toom-Cook4 algorithm once. It requires 3*3=9 64-degree polynomial multiplications using Karatsuba, and 7 using Toom-Cook4. There are 3 feasible methods using different operations shown in Table 2, and then we compare them to find the most efficient one.

$\boldsymbol{Method_A}$ We use Karatsuba algorithm of 2 recursive calls to split 256-degree polynomial into 64-degree, and then convert the low degree polynomials into big integers. We end up with 9 big integer multiplications. For the evaluation points of w_1, w_2, w_3, the inputs of w_1 and w_3 are the original 13-bit coefficients, but the

Table 2: Feasible methods of splitting 256 degree 13-bit coefficients polynomial multiplication

	Karatsuba [a]	Toom-Cook4[b]		
Evaluations	$3 \times \{w_1, w_2, w_3\}$	$\{w_1$ and $w_7\}$	$\{w_2$ to $w_6\}$	
Inputs	13-bit	13-bit	16-bit	
Operations	`KS1MUL`	`KS1MUL`	Karatsuba with `KS1MUL`	`KS2MUL`
Multiplications	9 2048 bits	2 2048 bits	5 × 3 1536 bits	5 × 2 1536 bits

[a] We define the method described in this column as $Method_A$ using Karatsuba algorithm of 2 recursive calls.

[b] We define $Method_B$ and $Method_C$ for the two types of operations for w_2 to w_6 respectively using Toom-Cook4.

inputs of w_2 are 14-bit (the sum of two 13-bit coefficients). We can compute w_1 and w_3 using `KS1MUL`(64,32) directly.

As the implementation result shown in Table 1, `KS1MUL` is faster than `KS2MUL`, but `KS2MUL` can allow longer bit length input. Since we compute the polynomial multiplication with the coefficients in $\mathbb{Z}_q(q = 2^{13})$ and there are only addition operations (no division) in the interpolation phase of Karatsuba, we can reduce all inputs to $\mathbb{Z}_q(q = 2^{13})$ before multiplication. For w_2, we can first reduce the inputs into 13 bits then also use `KS1MUL`(64,32). So using the Karatsuba algorithm for computing, a total of 9 `KS1MUL`(64,32) are required.

Method$_B$ Toom-Cook4 can split 256-degree polynomial multiplication into 7 64-degree, which is 2 low degree polynomial multiplications less than two recursive calls of Karatsuba, at the cost of more complicated computing in the interpolation phase. We use the 7 evaluation points of $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$ as same as the implementation of Saber in [16]. And following the Toom-Cook4 implementation in [16], in the interpolation phase, divisions by odd scalars are performed by computing multiplications by their respective inverses, and divisions by even scalars are performed in two steps: first multiply by the inverse of the odd factor, then compute a true division by the power-of-two factor. Hence, we need to add extra 3-bit precision for coefficients such that the extra bits can be used to calculate the divisions by 2, 4 and 8.

For the evaluation points w_1 and w_7, the inputs are the original 13-bit coefficients, so we use the efficient `KS1MUL`(64, 32). For the evaluation points w_2 to w_6, the inputs are the weighted sums of coefficients with bit length longer than 13-bit (i.e. $w_2 = (A_0 + 2A_1 + 4A_2 + 8A_3) * (B_0 + 2B_1 + 4B_2 + 8B_3)$). We must keep 13+3=16 bits of precision for their inputs, that is, we can only reduce the coefficients of inputs to 2^{16} before multiplication. We need to choose one more Karatsuba or `KS2MUL` for computing 16-bit coefficients inputs of w_2 to w_6, since `KS1MUL` only supports 13-bit coefficients.

In this method, we choose one more Karatsuba algorithm to compute w_2 to w_6. We split the 64-degree 16-bit coefficients polynomial multiplication into

3 32-degree, and pack one 16-bit coefficient into 16+16+log(32)=37 bits. We need to choose the 1536 bits multiplication as each polynomial is packed into a big integer of 37*32=1184 bits. To compute the entire 256-degree polynomial multiplication, the coprocessor is required to do 2 2048 bits multiplications (for computing w_1 and w_7 using `KS1MUL`) and 5*3=15 1536 bits multiplications (for computing w_2 to w_6 using one more Karatsuba and 3 `KS1MUL` each), for a total of 2*8.6K+5*3*5.0K=92.2K clock cycles. In $Method_A$, the coprocessor needs to do 9 2048 bits multiplication, a total of only 9 * 8.6K = 77.4K clock cycle. Moreover, the Toom-Cook4 of this method requires more complicated computation than the Karatsuba used in $Method_A$ in the interpolation phase, so this method is less efficient than $Method_A$.

***Method*$_C$** In this method, we use Toom-Cook4 and compute w_1 and w_7 as same as $Method_B$. Here we use `KS2MUL` to compute w_2 to w_6. We pack one 16-bit coefficient into (16+(1/2)*lg64)=19 bits and one 64 degree polynomial can be packed into 19*64=1216 bits. We still need to choose the 1536 bits multiplication of the coprocessor. Furthermore, we pack each coefficient into 24 bits (byte aligned) for more convenient packing operation, which can still be computed of 24*64=1536 bits multiplication.

To compute the entire 256-degree polynomial multiplication, the coprocessor is required to do 2 2048 bits multiplications (for computing w_1 and w_7 using `KS1MUL`) and 5*2=10 1536 bits multiplications (for computing w_2 to w_6 using `KS2MUL` including 2 1536 bits multiplications each), for a total of 2*8.6k+5*2*5.0k =67.2k clock cycles. This method is more efficient than $Method_A$ in terms of the cycles it takes for the coprocessor to perform the multiplications. However, the software-based functions, such as computing the value of the polynomial at positive and negative points, the addition and the division of big integers, lead to a large overhead in `KS2MUL`. To compute the entire 256 degree polynomial multiplication, it is required 2 `KS1MUL` and 5 `KS2MUL`, for a total of 2*10K+5*31K=175K clock cycles. However, in $Method_A$, 9 `KS1MUL` are required, for a total of 9*10k=90k clock cycles. Furthermore, Toom-Cook4 requires more computation than Karatsuba used in $Method_A$ in the interpolation phase, so this method is still less efficient than $Method_A$.

As a result, $Method_A$ is the most efficient one for 256-degree 13-bit coefficients polynomial multiplication.

256-degree 10-bit coefficients polynomial multiplication. For the 256-degree 10-bit coefficients polynomial multiplication, we consider using the Toom-Cook4 algorithm to split it into 7 64-degree. As mentioned earlier, we need to retain an additional 3 bits of precision in the Toom-Cook4 computation process. That is, we need to perform the polynomial multiplication of the 13-bit coefficients in the process. Here we can reduce all coefficients to 13 bits before multiplication then directly use the efficient algorithm `KS1MUL`(64, 32) for computation. Acually this is the most efficient method for 256-degree 10-bit coefficients polynomial multiplication.

Table 3: The most efficient methods to perform 256 degree polynomial multiplications in Saber

Coefficients	Modulus	Algorithm	Precision in progress	Kronecker	Multiplications
13-bit	8192	Karatsuba[a]	13 bits	`KS1MUL`	nine 2048 bits
10-bit	1024	Toom-Cook4	10+3=13 bits	`KS1MUL`	seven 2048 bits

[a] Two recursive calls.

4.2 Random generation

There is a true random number generator (TRNG) on ESP32. The true random numbers are generated based on the noise in the Wi-Fi/BT RF system, and can be read from the TRNG register of 32 bits. The TRNG is fed two bits of entropy every APB clock cycle of 80MHz. However the CPU is clocked at 240MHz, we are able to read the TRNG register at a maximum rate of 5 MHz for maximum amount of entropy. Hence at least 48 cycles should be waited between every two 32-bit random numbers read from the register. In our C implementation, when we read a 32-bit random number from the TRNG register, we need to unpack the 32-bit random number into 4 bytes and copy it to the target byte buffer. This unpacking operation requires 49 cycles, which is long enough for the TRNG to generate the next new 32-bit random number.

4.3 Using CPU idle time

In general, the Karatsuba and the Toom-Cook4 algorithms are composed of following phases: splitting, evaluation and interpolation. In the evaluation phase, we use the `KS1MUL` utilizing the big integer coprocessor to execute. We notice that the CPU is idle during the coprocessor execution. We propose the following strategies to make reasonable use of CPU idle time to improve the performance.

Pre-compute weighted sum of polynomials inputs. We choose to pre-compute weighted sum of polynomials inputs (i.e. $A(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3)$ for w_2 in Toom-Cook4) of the next `KSMUL` during the CPU idle time. For Karatsuba, we abandon the way of recursive calls to the Karatsuba algorithm and fully unrolled the recursively calls of Karatsuba algorithm. We can use this strategy for computing w_2 in Karatsuba and w_2 to w_6 in Toom-Cook4.

Rearrange interpolation steps into evaluation phase. We carefully rearrange the interpolation steps and maintain the correct execution order of the steps that have dependencies, such as the input of a step is the output of the previous step to ensure the correct result. For example, in the Toom-Cook4, we choose to compute w_1 to w_7 in the order of $\{w_1, w_3, w_4, w_5, w_2, w_7, w_6\}$. After w_3 and w_4 are completed, we compute the interpolation step of $w_4 = (w_4 - w_3)/2$ and $w_3 = w_3 + w_4$ during the CPU idle time while computing w_5 by the coprocessor. We note that we choose to compute w_1 first for the reason that, for w_1 there is no weighted sum of polynomials inputs are required to pre-compute.

Pre-algin inputs for writing to coprocessor registers. The process of performing big integer multiplication is as follows. To compute Z=X*Y, where X, Y, Z are big integers, CPU writes X and Y into the coprocessor's input data registers. When the execution of the coprocessor completes, CPU reads the value of Z from the output registers. The data registers of our coprocessor consist of several consecutive 32-bit registers. We first align the 64 16-bit elements to 64 32-bit consecutive data array, then use the `memcpy` function in standard C language for fast memory copying. We per-align the inputs for the coprocessor of the next `KSMUL` during the CPU idle time. This also avoids the overhead caused by the alignment of the data. In the first `KSMUL`, we use a `loop` function to write 16-bit elements into 32-bit registers one by one, since there is no CPU idle time to per-align. It takes 1207 clock cycles to write 256 data in a `loop`, but only 802 clock cycles to copy 64 consecutive 32-bit data using the `memcpy` function.

Post-reduce outputs read from coprocessor registers. We use the `memcpy` function to read the output from the coprocessor into a full buffer data array and choose to preform the reduction operation in the CPU idle time of the next `KSMUL`. It takes 1288 clock cycles to read 127 data one by one using a `loop` function, and it takes only 523 clock cycles to use the `memcpy` function for efficient memory copying.

We describe our implementations using these strategies in Algorithm 13 and Algorithm 14 in the appendix. As a result, for the entire polynomial multiplication, a large amount of software operations performed by CPU are arranged in its idle time during the co-processor computation, that is, the period between writing data (`write_regs`) to and reading data (`read_regs`) from the coprocessor registers. Since we ensure the CPU's executions can be completed before the coprocessor complete, the whole execution time mainly consists of the writing, computing, and reading time of the coprocessor, except for several remaining operations to process the final product.

4.4 Dual core acceleration

There are two CPU cores on ESP32 chip. The two cores are identical in function and share the same memory space. The address mappings of the two cores are symmetric, that is, accessing the same data variables using the same address.

We use FreeRTOS library [8] built in ESP32 for development work. FreeRTOS is a real-time operating system which provides APIs to execute computing task (code blocks) in a specified CPU core, as well as a semaphore mechanism that serves multi-core parallel computing. Here, the two cores are defined as "main core" and "secondary core". We control the entire algorithm flow in the main core while assigning some computing tasks to the secondary core. We execute the algorithm from the main core and initialize the variables. The two cores can access these variables with the same address and communicate based on the semaphore mechanism.

We hand-partition the task of Saber in order to achieve dual core parallel execution, since ESP32 does not have the ability to execute the single task on two cores based on hardware level instructions rearrangement. For the correctness and efficiency of the progress, we rearrange the small steps of Saber as following rules: if the input of a step does not depend on the output of the previous step, the two steps can be executed in parallel; otherwise we should execute this step after the previous step is completed. Figure explaining of dual-core task partitioning is described in Appendix J in the appendix.

The performance of dual core is twice that of a single core in the ideal state. However, the dual-core parallel execution of the Saber algorithm has two main limitations. First, Saber itself is a sequential execution algorithm, and the input to many steps depends on the results of the previous step, so we can't parallel all the steps. Also, ESP32 has only one big integer coprocessor and one TRNG, and we can only perform polynomial multiplication and random number generation operations in one core.

We note that our dual-core acceleration is valuable for decreasing latency of the common applications running on ESP32 as IoT clients to communicate with remote server via "PQC-TLS" based network connection. Also there are few requirements for network throughput on these IoT clients.

4.5 Generation of the matrix A

In the `GEN` algorithm, we choose to generate the vector $\boldsymbol{s}$ in the main core, generate the matrix $\boldsymbol{A}$ in the secondary core, and then compute the product of the two in the main core. It takes 117K clock cycles to generate $\boldsymbol{s}$ and 477K to generate $\boldsymbol{A}$. Hence the main core needs to wait for the secondary core to complete before performing the next multiplication of $\boldsymbol{A}$ and $\boldsymbol{s}$. To reduce the wait time, the main core can begin to perform the multiplication when the secondary core generates a row of elements (or even an element) in $\boldsymbol{A}$. In original implementation of Saber, $\boldsymbol{A}$ was generated in two steps: using the `shake128` algorithm to stretch a 32-byte seed into a full buffer of bytes sufficient to generate 9 polynomials at a time, and then convert the bytes into 9 polynomials. The `shake128` takes up 89% in the process of generating $\boldsymbol{A}$, so we need to "split" the `shake128` function to output bytes segment by segment and convert the byte segments to polynomials one by one. We use the idea of the *justintime* strategy [22], which was originally used to save memory in implementing Saber on ARM, to generate 9 polynomials in matrix $\boldsymbol{A}$ one by one, and use semaphores to inform the main core when a polynomial in matrix $\boldsymbol{A}$ is generated in the secondary core. In our implementation, we remove the global variables of the polynomial and byte count information in the original implementation of *justintime*, and add the appropriate semaphore mechanism for parallelization.

Compared to the original implementation, there is some additional overhead in *justintime*, since it needs to handle the "leftover" bits and count the bytes length. In our implementation, the modified *justintime* is only used in the `GEN` algorithm to reduce the wait time before the multiplication of $\boldsymbol{A}$ and $\boldsymbol{s}$. The original version is used in `ENC` and `DEC` since we have sufficient time in the secondary core to pre-generate $\boldsymbol{A}$ before $\boldsymbol{A}$ and $\boldsymbol{s}$ are multiplied by rearranging the steps.

5 Results

5.1 Implementation performance

We develop our implementation in C language, based on the SDK for ESP32 provided in [7]. We execute the implementation on an ESP32-DevKitC development board [6] to evaluate the performance. And we use the official function `ESP.getCycleCount()` to count the clock cycles.

Table 4: Performance of 256 degree polynomial multiplication on ESP32

Implementation	Modulus	Cycles
NTT[a]	7681	243,967
Toom-Cook4 with `KS1MUL` [b]	7681	127,293
Toom-Cook4 with `KS1MUL` (parallel) [c]	7681	94,537
Karatsuba with `KS1MUL`	8192	130,025
Karatsuba with `KS1MUL` (parallel)	8192	97,050
Toom-Cook4 with `KS1MUL`	1024	105,633
Toom-Cook4 with `KS1MUL` (parallel)	1024	85,178

[a] Based on the NTT implementation of Kyber submission [27]. Including 2*Forward NTT + 1*Pointwise Multiplication + 1*Inverse NTT.

[b] No extra bit precision acquired in the interpolation phase in Toom-Cook4 since all the divisions are performed by computing multiplications by their respective inverses with the prime modulus ($q = 7681$). The coefficients are packed into 32 bits.

[c] The "parallel" version is running CPU and coprocessor in parallel with "using CPU idle time" strategies.

In Table 4, we compare the performance of a single 256-degree polynomial multiplication with different modulus on ESP32. Here we use the source code in Kyber [14] [27] to execute the multiplication by NTT, and execute the implementations we presented in previous sections utilizing the big integer coprocessor. We note that the cycles in Table 4 includes the reduction. The reduction is performed as Algorithm 10 in the appendix except NTT-based multiplication and the reduction operation is cheap. The correctness of these implementations in Table 4 has been checked and the source code of these implementations is also available in our github link.

As can be seen from the Table 4, our implementations utilizing the big integer coprocessor are faster than NTT-based multiplication with modulus 7681 by around 2.6x, 2.5x and 2.9x times with moduli 7681, 8192, 1024 respectively. It can also be seen that by using CPU idle time to allow the CPU and the coprocessor to run in parallel, the clock cycles of 28.7%, 25.4%, 19.4% (parallel versions in the table) can be reduced, respectively.

Table 5: Performance of polynomial multiplication functions in Saber on ESP32

Functions	Cycles
MatrixMulRounding[a]	827,050
VectorMul	243,023

[a] Merged operation of `MatrixMul` and `Rounding`.

Table 5 shows the clock cycles of polynomial multiplication functions in Saber. The benefit of using CPU idle time strategies is also appreciable in `MatrixMulRounding` and `VectorMul`, where the total clock cycles are smaller than the times count of single polynomial multiplication (for `MatrixMulRounding`, 827,050 vs. 3*3*97,050=873,450; for `VectorMul`, 243,023 vs. 3*85,178=255,534).

Table 6: Performance of Saber on ESP32

Implementation	Algorithm	Cycles	Run time(ms)	Speedup ratio
Reference [15]	`CCA.GEN`	12,287,254	51.2	1x
	`CCA.ENC`	16,365,828	68.2	1x
	`CCA.DEC`	20,042,134	83.5	1x
ESP32 (single-core)	`CCA.GEN`	1,638,677	6.8	7.5x
	`CCA.ENC`	2,123,010	8.8	7.7x
	`CCA.DEC`	2,192,991	9.1	9.1x
ESP32 (dual-core)	`CCA.GEN`	1,176,191	4.9	10.4x
	`CCA.ENC`	1,624,650	6.8	10.1x
	`CCA.DEC`	1,514,185	6.3	13.2x

The performance of our implementation for Saber on ESP32 are listed in Table 6. Our single-core version implementation is faster than the reference by 7.5x, 7.7x and 9.1x times for `GEN`, `ENC`, `DEC` respectively, and the dual-core version is faster than the reference by 10.5x, 10.1x and 13.2x times for `GEN`, `ENC`, `DEC` respectively.

We note that we use the Saber round-1 submission [15] as the "Reference" in Table 6 and our optimized implementation is based on it, for making a direct comparison with previous optimized implementation [20] with the same "Reference" in the next sub-section. Also the changes in the Saber round-2 submission [17] listed in its supporting documentation make a negligible difference on the performance compared with round-1: "Transposing matrix $\boldsymbol{A}$" has no impact of performance in our optimized implementation since our performance of polynomial multiplication is irrelative of transposing; "The parameter T" and "Simplification of the specification" have no impact on the implementation; "Replacement of constant polynomial h" slightly changes the implementation and has negligible impact on the actual performance.

5.2 Comparison with related work

Table 7: Performance of Saber on Cortex-M4[20]

Implementation	Algorithm	Cycles	Run time(ms)	Speedup ratio
Reference [15] [a]	`CCA.GEN`	6,530,000	40.8	1x
	`CCA.ENC`	8,684,000	54.3	1x
	`CCA.DEC`	10,581,000	66.1	1x
Cortex-M4 [20]	`CCA.GEN`	895,000	5.6	7.3x
	`CCA.ENC`	1,161,000	7.3	7.5x
	`CCA.DEC`	1,204,000	7.5	8.8x

[a] The cycles are reported in [20].

Cortex-M4, belongs to ARM Cortex-M series, has been well-studied for implementing post-quantum cryptography. The performance of the fastest optimized implementation of Saber from [20] is shown in Table 7. It is not fair to compare the cycles of the same cryptography scheme in these two different platforms, since Cortex-M4 with an ARM CPU and ESP32 with Xtensa LX6 CPUs are of different CPU families with different instruction sets. But as can be seen from two tables, with the "Reference" [15], the speedup ratios of our single-core implementations on ESP32 are slightly higher than the speedup ratios of [20] on Cortex-M4, and our dual-core version are even higher. On the other hand, the run time of our dual-core version is better than [20].

Table 8: Comparison of HW and SW functions for polynomial multiplications

Implementation	Algorithm	Type	Functions	Cycles	Percentage
Kyber variant [10]	`CPA.GEN`	HW	`MulAddSingle`, `FinalEll`[a]	1,901,046	58%
		SW	`Snort`, `Sneeze`	1,352,445	42%
	`CPA.ENC`	HW	`MulAddSingle`, `FinalEll`	2,534,728	59%
		SW	`Snort`, `Sneeze`	1,772,243	41%
	`CPA.DEC`	HW	`MulAddSingle`, `FinalEll`	633,682	55%
		SW	`Snort`, `Sneeze`	512,849	45%
Saber (ours)	`CPA.GEN`	HW	`MatrixMulRounding`	695,547	84%
		SW	Internal operations	131,503	16%
	`CPA.ENC`	HW	`MatrixMulRounding`, `VectorMul`	875,874	82%
		SW	Internal operations	194,199	18%
	`CPA.DEC`	HW	`VectorMul`	180,327	74%
		SW	Internal operations	62,696	26%

[a] The detail cycles of internal sw-based operations are not reported.

In [10], Albrecht et al. presented an implementation of Kyber variant utilizing the RSA coprocessor (big integer coprocessor) on SLE78. We emphasize that our approach is different from the one used in [10]. Albrecht et al. split the **ring** with the idea from Schönhage [26] or Nussbaumer [24] and computed the polynomial multiplication $C(x) = A(x) * B(x) \bmod^{+} F$ for $A, B, C \in \mathbb{Z}_p$ with $p = F = 2^{2048} + 1$ (converted to big integer **modular multiplications**) by the RSA coprocessor on SLE78, meanwhile we split the **polynomial multiplication** and computed the small degree polynomial multiplications (converted to big integer **standard multiplications**) by the big integer coprocessor on ESP32.

For polynomial multiplication, Kyber(n=256, q=7681, k=3) and Saber(n=256, q=8192 or 1024, k=3) are of similar parameters. For CPA secure `GEN`, `ENC`, `DEC` of functions for polynomial multiplication, implementation of [10] costs 81, 108, 27 calls of hardware-based 2048+1 bits **modular multiplications** in `MulAddSingle` respectively, while ours costs 81, 102, 21 calls of hardware-based 2048 bits **standard multiplications**.

With similar computational complexity of hardware-based functions (ours may be less complex since computing standard multiplication is cheaper than the similar bits of modular multiplication and there are also 3 multiplications in one call of `FinalEll`), due to use the coprocessor, the implementation in [10] has a large amount of software overhead. There are 42%, 41%, 45% of total cycles cost by additional software functions of CPA-secure `GEN`, `ENC`, `DEC` respectively. As a result, our approach takes a better advantage of the big integer coprocessor with significantly lower software overhead.

We note that the cycles of [10] in Table 8 are computed by the cycles of single function and the number of calls of KS1 version reported from its Table 3. Although the cycles of KS2 version (slightly less calls of hardware-based modular multiplication but similar total cycles) is slightly different, we can still get the same comparison result.

6 Conclusion

In this paper, we provide an efficient implementation of polynomial multiplications and a speed-optimized implementation of the CCA-secure lattice-based key encapsulation scheme Saber on embedded microcontroller ESP32.

The efficient implementation of polynomial multiplications utilizing the big integer coprocessor outperforms the NTT-based multiplications on our platform, and also contains significantly lower software overhead than the implementation of [10]. Our fastest dual-core version implementation of Saber takes 1176K, 1625K, 1514K clock cycles for key generation, encapsulation and decapsulation respectively, that is, 4.9, 6.8, 6.3 milliseconds assuming 240 MHz frequency to execute. We have shown that the existing big integer coprocessor originally designed for the acceleration of RSA or ECC is available for making a significant speedup for the time-consuming polynomial multiplications in lattice-based cryptography. The dual core is also a good feature to get a better performance for the scheme designed of sequential execution, when properly assigning tasks to two cores running in parallel.

References

1. National institute of standards and technology. submission requirements and evaluation criteria for the post-quantum cryptography standardization process. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf (2016)
2. NIST post-quantum cryptography round 1 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (2017)
3. TSR Report: 2017 wireless connectivity market analysis. www.t-s-r.co.jp/e/report/4543.html (2018)
4. Espressif milestones. www.espressif.com/en/company/about-us/milestones (2019)
5. mbedtls. https://tls.mbed.org/ (2019)
6. ESP32 development-boards. https://www.espressif.com/en/products/hardware/development-boards (2019)
7. ESP32 software development kit. https://github.com/espressif/arduino-esp32 (2019)
8. FreeRTOS. https://www.freertos.org/ (2019)
9. NIST post-quantum cryptography round 2 submissions. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions (2019)
10. Albrecht, M.R., Hanser, C., Höller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Implementing rlwe-based schemes using an RSA co-processor. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(1), 169–208 (2019), https://doi.org/10.13154/tches.v2019.i1.169-208
11. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 327–343. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim
12. Bodrato, M., Zanoni, A.: Integer and polynomial multiplication: towards optimal toom-cook matrices. In: Wang, D. (ed.) Symbolic and Algebraic Computation, International Symposium, ISSAC 2007, Waterloo, Ontario, Canada, July 28 - August 1, 2007, Proceedings. pp. 17–24. ACM (2007), https://doi.org/10.1145/1277548.1277552
13. Bos, J.W., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., Stebila, D.: Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1006–1018. ACM (2016), https://doi.org/10.1145/2976749.2978425
14. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 353–367. IEEE (2018), https://doi.org/10.1109/EuroSP.2018.00032
15. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber algorithm information in the NIST round-1 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (2017)
16. D'Anvers, J., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure KEM. In: Joux, A., Nitaj, A.,

Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10831, pp. 282–305. Springer (2018), https://doi.org/10.1007/978-3-319-89339-6_16

17. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber algorithm information in the NIST round-2 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions (2019)
18. Harvey, D.: Faster polynomial multiplication via multipoint kronecker substitution. J. Symb. Comput. **44**(10), 1502–1510 (2009), https://doi.org/10.1016/j.jsc.2009.05.004
19. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the fujisaki-okamoto transformation. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10677, pp. 341–371. Springer (2017), https://doi.org/10.1007/978-3-319-70500-2_12
20. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11464, pp. 281–301. Springer (2019), https://doi.org/10.1007/978-3-030-21568-2_14
21. Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk. vol. 145, pp. 293–294. Russian Academy of Sciences (1962)
22. Karmakar, A., Mera, J.M.B., Roy, S.S., Verbauwhede, I.: Saber on ARM cca-secure module lattice-based key encapsulation on ARM. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 243–266 (2018), https://doi.org/10.13154/tches.v2018.i3.243-266
23. Knuth, D.E.: The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition. Addison-Wesley (1997), http://www.worldcat.org/oclc/312910844
24. Nussbaumer, H.: Fast polynomial transform algorithms for digital convolution. IEEE Transactions on Acoustics, Speech, and Signal Processing **28**(2), 205–215 (1980)
25. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. arXiv preprint quant-ph/0301141 (2003)
26. Schönhage, A.: Schnelle multiplikation von polynomen über körpern der charakteristik 2. Acta Inf. **7**, 395–398 (1977), https://doi.org/10.1007/BF00289470
27. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehle, D.: Kyber algorithm information in the NIST round-2 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (2019)
28. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997), https://doi.org/10.1137/S0097539795293172

A CPA secure Saber KEM

Algorithm 5: `Saber.KeyGen()`

```
$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} \leftarrow \mathtt{gen}(seed_{\boldsymbol{A}}) \in R_q^{l\times l}$
$\boldsymbol{s} \leftarrow \beta_\mu(R_q^{l\times 1})$
$\boldsymbol{b} = \mathtt{bits}(\boldsymbol{As} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_q^{l\times 1}$
return $(pk := (\boldsymbol{b}, seed_{\boldsymbol{A}}), sk := \boldsymbol{s})$
```

Algorithm 6: `Saber.Enc`$(pk = (b, seed_A), m \in M; r)$

```
$\boldsymbol{A} \leftarrow \mathtt{gen}(seed_{\boldsymbol{A}}) \in R_q^{l\times l}$
$\boldsymbol{s'} \leftarrow \beta_\mu(R_q^{l\times 1})$
$\boldsymbol{b'} = \mathtt{bits}(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l\times 1}$
$v' = \boldsymbol{b}^T\mathtt{bits}(\boldsymbol{s'}, \epsilon_p, \epsilon_p) + h_1 \in R_p$
$c_m = \mathtt{bits}(v' + 2^{\epsilon_p - 1}m, \epsilon_p, \epsilon_t + 1) \in R_{2t}$
return $c := (c_m, \boldsymbol{b'})$
```

Algorithm 7: `Saber.Dec`$(sk = \boldsymbol{s}, c_m, \boldsymbol{b'})$

```
$v = \boldsymbol{b'}^T\mathtt{bits}(\boldsymbol{s}, \boldsymbol{\epsilon_p}, \boldsymbol{\epsilon_p}) + h_1 \in R_p$
$m' = \mathtt{bits}(v - 2^{\epsilon_p - \epsilon_t - 1}c_m + h_2, \epsilon_p, 1) \in R_2$
return $m'$
```

B CCA secure Saber KEM

Algorithm 8: $\texttt{Saber.Encaps}(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}))$

```
$m \leftarrow \mathcal{U}\{0,1\}^{256}$
$(\hat{K}, r) = \mathcal{G}(pk, m)$
$c = \texttt{Saber.Enc}(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
return $(c, K)$
```

Algorithm 9: $\texttt{Saber.Decaps}(sk = (\boldsymbol{s}, z), pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}, c))$

```
$m' = \texttt{Saber.Dec}(\boldsymbol{s}, c)$
$(\hat{K}', r') = \mathcal{G}(pk, m')$
$c' = \texttt{Saber.Enc}(pk, m'; r')$
if $c = c'$ then
    return $K = \mathcal{H}(\hat{K}', c)$
else
    return $K = \mathcal{H}(z, c)$
```

C Performance of big integer multiplication by coprocessor

Table 9: Performance of big integer multiplication by coprocessor

Bits	Cycles [a]
1536	4,966
2048	8,587

[a] Only including the cycles of computing. (Not including registers writing and reading)

D Reduction in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$

Algorithm 10: `Reduction`(h, n, q)

```
Input: product $h$ of two $n$ degree polynomials
Input: modulus $q$ of coefficients in $h$ to reduce
Output: reduced polynomial $r$ from $h$ in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$
for $i = 0, 1, \dots, n-2$ do
    $r_i \leftarrow (h_i - h_{(i+n)}) \bmod q$
$r_{(n-1)} \leftarrow h_{(n-1)} \bmod q$
return $r$;
```

E Performance of other key functions in Saber on ESP32

Table 10: Performance of other key functions in Saber on ESP32

Functions	Cycles
Gen($\boldsymbol{A}$)	476,658
shake128(32B→3744B) [a]	430,184
Gen($\boldsymbol{s}$)	116,548
shake128(32B→768B)[b]	102,950
sha256(1088B)	210,643
sha256(992B)	183,213
sha256(64B)	27,129
sha512(64B)	26,365

[a] Internal operation in Gen($\boldsymbol{A}$).

[b] Internal operation in Gen($\boldsymbol{s}$).

F Karatsuba Algorithm

Algorithm 11: Karatsuba Algorithm

```
Input: Two polynomials $A(x)$ and $B(x)$ of degree $n$
Output: $C(x) = A(x) * B(x)$
// Splitting $A(x)$ and $B(x)$ into two polynomials of size $\frac{n}{2}$
$A(y) = A_1 \cdot y + A_0$ where $y = x^{\frac{n}{2}}$
$B(y) = B_1 \cdot y + B_0$
// Evaluation of the Polynomials at $y = \{0, 1, \infty\}$
$w_1 = A(\infty) * B(\infty) = A_1 * B_1$
$w_2 = A(1) * B(1) = (A_0 + A_1) * (B_0 + B_1)$
$w_3 = A(0) * B(0) = A_0 * B_0$
// Interpolation
$w_2 = w_2 - w_1 - w_3$
return $w_1 \cdot y^2 + w_2 \cdot y + w_3$
```

G Toom-Cook4 Algorithm

Algorithm 12: Toom-Cook4 Algorithm

```
Input: Two polynomials $A(x)$ and $B(x)$ of degree $n$
Output: $C(x) = A(x) * B(x)$
// Splitting $A(x)$ and $B(x)$ into four polynomials of size $\frac{n}{4}$
$A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{\frac{n}{4}}$
$B(y) = B_3 \cdot y^3 + B_2 \cdot y^2 + B_1 \cdot y + B_0$
// Evaluation of the Polynomials at $y = \{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$
$w_1 = A(\infty) * B(\infty) = A_3 * B_3$
$w_2 = A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3)$
$w_3 = A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3)$
$w_4 = A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3)$
$w_5 = A(\frac{1}{2}) * B(\frac{1}{2}) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3)$
$w_6 = A(\frac{-1}{2}) * B(\frac{-1}{2}) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3)$
$w_7 = A(0) * B(0) = A_0 * B_0$
// Interpolation
$w_2 = w_2 + w_5$
$w_6 = w_6 - w_5$
$w_4 = (w_4 - w_3)/2$
$w_2 = w_5 - w_1 - 64 \cdot w_7$
$w_3 = w_3 + w_4$
$w_5 = 2 \cdot w_5 + w_6$
$w_2 = w_2 - 65 \cdot w_3$
$w_3 = w_3 - w_7 - w_1$
$w_2 = w_2 + 45 \cdot w_3$
$w_5 = (w_5 - 8 \cdot w_3)/24$
$w_6 = w_6 + w_2$
$w_2 = (w_2 + 16 \cdot w_4)/18$
$w_3 = w_3 - w_5$
$w_4 = -(w_4 + w_2)$
$w_6 = (30 \cdot w_2 - w_6)/60$
$w_2 = w_2 - w_6$
return $w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7$
```

H Implementation of unrolled Karatsuba with KS1MUL

Algorithm 13: Unrolled Karatsuba parallel

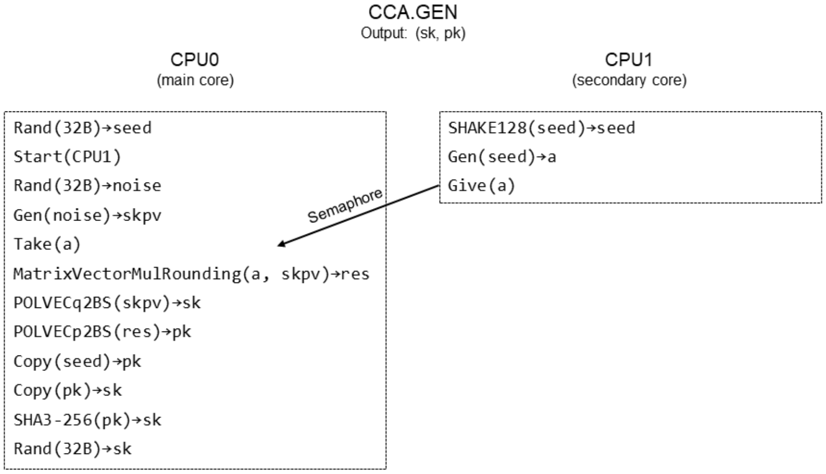
```
Input: polynomials $a, b \in \mathbf{Z}[x]$ with same degree $n = 256$
Output: $root\_res = a * b$
$aL = \&a[0]$; $aH = \&a[128]$; $bL = \&b[0]$; $bH = \&b[128]$;
$aLL = \&a[0]$; $aLH = \&a[64]$; $aHL = \&a[64*2]$; $aHH = \&a[64*3]$;
$bLL = \&b[0]$; $bLH = \&b[64]$ ; $bHL = \&b[64*2]$; $bHH = \&b[64*3]$;
// $root\_w3 \leftarrow (aL, bL)$
write_regs($aLL, bLL$)                         // write aLL, bLL by loop function
$aS = aL + aH$; $bS = bL + bH$;
$aSL = \&aS[0]$; $aSH = \&aS[64]$; $bSL = \&bS[0]$; $bSH = \&bS[64]$;
$aREG$ = alignment($aLH$); $bREG$ = alignment($bLH$);
read_regs($rREG$);
write_regs($aREG, bREG$)                       // write aLH, bLH by memcpy function
$sub\_w3$ = reduction($rREG$)                          // reduce 32-bit to 13-bit
$aLS = aLL + aLH$; $bLS = bLL + bLH$
$aREG$ = alignment($aLS$); $bREG$ = alignment($bLS$);            // 32-bit alignment
read_regs($rREG$);
write_regs($aREG, bREG$)                                                // aLS,bLS
$sub\_w1$ = reduction($rREG$);
$aREG$ = alignment($aHL$); $bREG$ = alignment($aHL$);
read_regs($rREG$);
// $root\_w1 \leftarrow (aH, bH)$
write_regs($aREG, bREG$) ;                                              // aHL,bHL
$sub\_w2$ = reduction($rREG$); $sub\_w2 = sub\_w2 - sub\_w1 - sub\_w3$ ;
$root\_w3$ = recomposition($sub\_w1, sub\_w2, sub\_w3$);
$aREG$ = alignment($aHH$); $bREG$ = alignment($aHH$);
read_regs($rREG$);
write_regs($aREG, bREG$)                                                // aHH,bHH
$sub\_w3$ = reduction($rREG$);
$aHS = aHL + aHH$; $bHS = bHL + bHH$;
$aREG$ = alignment($aHS$); $bREG$ = alignment($bHS$);
read_regs($rREG$);
write_regs($aREG, bREG$);                                               // aHS,bHS
$sub\_w1$ = reduction($rREG$)
$aREG$ = alignment($aSL$); $bREG$ = alignment($bSL$);
read_regs($rREG$);
// $root\_w2 \leftarrow (aS, bS)$
write_regs($aREG, bREG$);                                               // aSL,bSL
$sub\_w2$ = reduction($rREG$);
$sub\_w2 = sub\_w2 - sub\_w1 - sub\_w3$ ;
$root\_w1$ = recomposition($sub\_w1, sub\_w2, sub\_w3$);
$aREG$ = alignment($aSH$); $bREG$ = alignment($bSH$);
read_regs($rREG$);
write_regs($aREG, bREG$)                                                // aSH,bSH
$sub\_w3$ = reduction($rREG$);
$aSS = aSL + aSH$; $bSS = bSL + bSH$
$aREG$ = alignment($aSS$); $bREG$ = alignment($bSS$);
read_regs($rREG$);
write_regs($aREG, bREG$);                                               // aSS,bSS
$sub\_w1$ = reduction($rREG$);
read_regs($rREG$);
$sub\_w2$ = reduction($rREG$);
$sub\_w2 = sub\_w2 - sub\_w1 - sub\_w3$ ;
$root\_w2$ = recomposition($sub\_w1, sub\_w2, sub\_w3$);
$root\_w2 = root\_w2 - root\_w1 - root\_w3$;
$root\_res$ = recomposition($root\_w1, root\_w2, root\_w3$);
return reduction($root\_res$);
```

I Implementation of Toom-Cook4 with KS1MUL

Algorithm 14: Toom-Cook4 parallel

```
Input: polynomials $a, b \in \mathbf{Z}[x]$ with same degree $n = 256$
Output: $res = a * b$
$a_0 = \&a[0]$; $a_1 = \&a[64]$; $a_2 = \&a[64 * 2]$; $a_3 = \&a[64 * 3]$;
$b_0 = \&b[0]$; $b_1 = \&b[64]$; $b_2 = \&b[64 * 2]$; $b_3 = \&b[64 * 3]$;
$ainf = a_3$; $binf = b_3$;
// $w1 \leftarrow ainf * binf$
write_regs($ainf, binf$);                                    // write ainf,binf by loop
$a1 = a_0 + a_1 + a_2 + a_3$; $b1 = b_0 + b_1 + b_2 + b_3$ ;
$an1 = a_0 + a_2 - (a_1 + a_3)$; $bn1 = b_0 + b_2 - (b_1 + b_3)$;
$aREG$ = alignment($a1$); $bREG$ = alignment($b1$);                // 32-bit alignment
read_regs($rREG$);
// $w3 \leftarrow a1 * b1$
write_regs($aREG, bREG$);                                     // write a1,b1 by memcpy
$w1$ = reduction($rREG$);
$ahalf = 8 * a_0 + 4 * a_1 + 2 * a_2 + a_3$; $bhalf = 8 * b_0 + 4 * b_1 + 2 * b_2 + b_3$;
$anhalf = 8 * a_0 + 2 * a_2 - (4 * a_1 + a_3)$; $bnhalf = 8 * b_0 + 2 * b_2 - (4 * b_1 + b_3)$;
$aREG$ = alignment($an1$); $bREG$ = alignment($bn1$);
read_regs($rREG$);
// $w4 \leftarrow an1 * bn1$
write_regs($aREG, bREG$);                                                // an1,bn1
$w3$ = reduction($rREG$);
$a2 = a_0 + 2 * a_1 + 4 * a_2 + 8 * a_3$; $b2 = b_0 + 2 * b_1 + 4 * b_2 + 8 * b_3$;
$aREG$ = alignment($ahalf$); $bREG$ = alignment($bhalf$);
read_regs($rREG$);
// $w5 \leftarrow ahalf * bhalf$
write_regs($aREG, bREG$);                                            // ahalf,bhalf
$w4$ = reduction($rREG$);
$w4 = (w4 - w3)/2$;
$w3 = w3 + w4$;
$aREG$ = alignment($a2$); $bREG$ = alignment($b2$);
read_regs($rREG$);
// $w2 \leftarrow a2 * b2$
write_regs($aREG, bREG$);                                                  // a2,b2
$w5$ = reduction($rREG$);
$w5\_copy = w5$;
$w5 = w5 - w1$;
$aREG$ = alignment($a0$); $bREG$ = alignment($b0$);
read_regs($rREG$);
// $w7 \leftarrow a0 * b0$
write_regs($aREG, bREG$)                                                   // a0,b0
$w2$ = reduction($rREG$);
$w2 = w2 + w5\_copy$;
$w2 = w2 - 65 * w3$;
$w3 = w3 - w1$;
$aREG$ = alignment($anhalf$); $bREG$ = alignment($bnhalf$);
read_regs($rREG$);
// $w6 = anhalf * bnhalf$
write_regs($aREG, bREG$)                                           // anhalf,bnhalf
$w7$ = reduction($rREG$);
$w5 = w5 - 64 * w7$; $w3 = w3 - w7$;
$w2 = w2 + 45 * w3$; $w2\_copy = w2$;
$w2 = (w2 + 16 * w4)/18$; $w4 = -(w4 + w2/2)$;
read_regs($rREG$);
$w7$ = reduction($rREG$);
$w6 = w6 - w5\_copy$; $w5 = 2 * w5 + w6$;
$w5 = (w5 - 8 * w3)/24$; $w6 = w6 + w2\_copy$;
$w3 = w3 - w5$; $w6 = (30 * w2 - w6)/60$; $w2 = w2 - w6$;
$res$ = recomposition($w_1, w_2, w_3, w_4, w_5, w_6, w7$);
return reduction($res$)
```

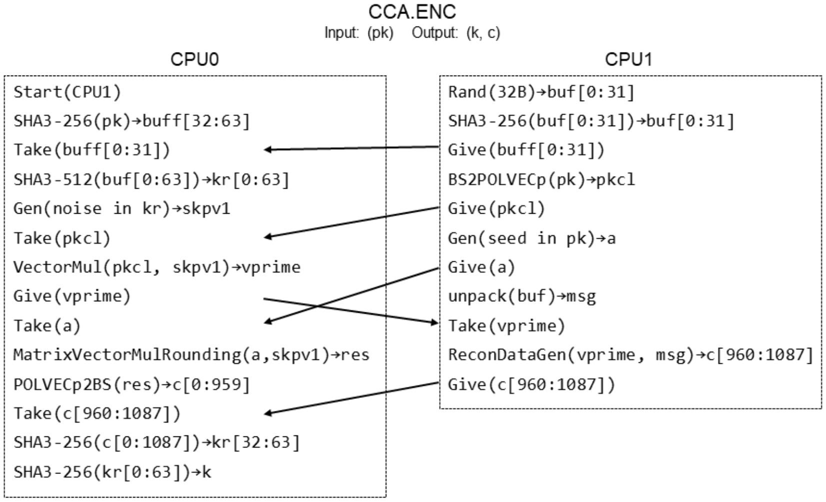
J Figure explaining of dual-core task partitioning



```
Rand(32B)→seed
Start(CPU1)
Rand(32B)→noise
Gen(noise)→skpv
Take(a)
MatrixVectorMulRounding(a, skpv)→res
POLVECq2BS(skpv)→sk
POLVECp2BS(res)→pk
Copy(seed)→pk
Copy(pk)→sk
SHA3-256(pk)→sk
Rand(32B)→sk
```

```
SHAKE128(seed)→seed
Gen(seed)→a
Give(a)
```

Fig. 1: Task partitioning of CCA secure GEN. The arrow indicates the semaphore for two cores communication.



```
Start(CPU1)
SHA3-256(pk)→buff[32:63]
Take(buff[0:31])
SHA3-512(buf[0:63])→kr[0:63]
Gen(noise in kr)→skpv1
Take(pkcl)
VectorMul(pkcl, skpv1)→vprime
Give(vprime)
Take(a)
MatrixVectorMulRounding(a,skpv1)→res
POLVECp2BS(res)→c[0:959]
Take(c[960:1087])
SHA3-256(c[0:1087])→kr[32:63]
SHA3-256(kr[0:63])→k
```

```
Rand(32B)→buf[0:31]
SHA3-256(buf[0:31])→buf[0:31]
Give(buff[0:31])
BS2POLVECp(pk)→pkcl
Give(pkcl)
Gen(seed in pk)→a
Give(a)
unpack(buf)→msg
Take(vprime)
ReconDataGen(vprime, msg)→c[960:1087]
Give(c[960:1087])
```

Fig. 2: Task partitioning of CCA secure ENC

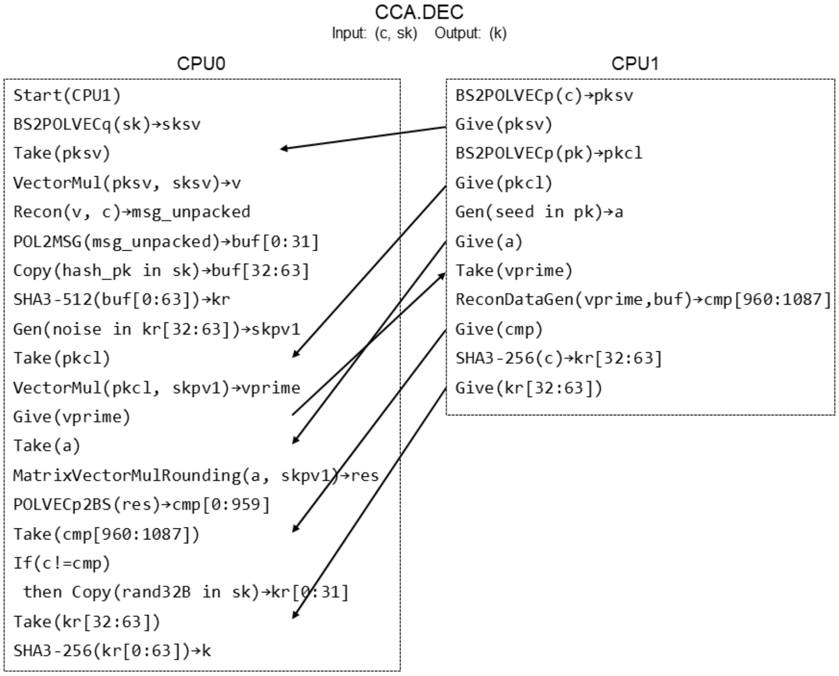


Fig. 3: Task partitioning of CCA secure DEC