Saber on ARM

CCA-secure module lattice-based key encapsulation on ARM

Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy and Ingrid Verbauwhede

imec-COSIC, KU Leuven
Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium
{firstname.lastname}@esat.kuleuven.be

Abstract. The CCA-secure lattice-based post-quantum key encapsulation scheme Saber is a candidate in the NIST's post-quantum cryptography standardization process. In this paper, we study the implementation aspects of Saber in resource-constrained microcontrollers from the ARM Cortex-M series which are very popular for realizing IoT applications. In this work, we carefully optimize various parts of Saber for speed and memory. We exploit digital signal processing instructions and efficient memory access for a fast implementation of polynomial multiplication. We also use memory efficient Karatsuba and just-in-time strategy for generating the public matrix of the module lattice to reduce the memory footprint. We also show that our optimizations can be combined with each other seamlessly to provide various speed-memory trade-offs. Our speed optimized software takes just 1,147K, 1,444K, and 1,543K clock cycles on a Cortex-M4 platform for key generation, encapsulation and decapsulation respectively. Our memory efficient software takes 4,786K, 6,328K, and 7,509K clock cycles on an ultra resource-constrained Cortex-M0 platform for key generation, encapsulation, and decapsulation respectively while consuming only 6.2 KB of memory at most. These results show that lattice-based key encapsulation schemes are perfectly practical for securing IoT devices from quantum computing attacks.

Keywords: Key encapsulation scheme, post-quantum cryptography, lattice-based cryptography, efficient software, Saber

Introduction

In as early as 1981, famous physicist Richard Feynman in a lecture titled 'Simulating physics with computers' [21] delineated the use of *Superposition principle* of quantum mechanics to build powerful computers. This is widely considered as laying the bedrock of modern quantum computers. While a powerful quantum computer opens the door for multitude of philanthropic activities like accelerated drug search for various diseases, DNA sequencing, powerful artificial intelligence etc., we cannot ignore the other side of the coin where an entity utilizes these massively powerful computers for malicious use. Especially, security of our current public-key infrastructure is highly vulnerable as the two most widely used hard problems in public-key cryptography namely integer factorization and elliptic curve discrete logarithm can be solved with a quantum computer running Shor's [40] and Proost and Zalka's [35] algorithm respectively. The most recent breakthrough came in March, 2018 when google announced its 72-qubit quantum computer [28]. Though this is too small to pose any real danger, the research and development in quantum computing is advancing at a rapid scale.

Post-quantum cryptography is a branch of cryptography that focuses on designing schemes that are secure against quantum computing attacks. Recently, several hard problems from the lattice theory have become popular for constructing post-quantum public-key cryptographic schemes. These problems are presumed to be computationally extremely hard even for a powerful quantum computer. In 2016, the National Institute of Standards and Technology (NIST) recommended a gradual shift towards post-quantum cryptography and took a prudent step by calling for proposals [16] for standardizing post-quantum cryptography protocols to maintain the security of our digital world in the event of the arrival of quantum computers. Incidentally, a large number of proposals submitted in the NIST's standardization process are based on hard problems over lattices. Lattice-based cryptography offers wide applicability, computational efficiency, strong security, and relatively shorter key sizes; yet its real deployment in a wide variety of computing devices and applications faces several challenges. One such platform is the resource constrained microcontrollers such as ARM Cortex-M0 and Cortex-M4. These two platforms are very popular for realizing Internet of Things (IoT) applications. The IoT devices are increasingly infiltrating our daily lives and it is imperative to secure these devices from a wide range of attacks.

Our paper is aimed at implementing the chosen ciphertext attack (CCA) resistant lattice-based key encapsulation mechanism (KEM) Saber [17] on resource-constrained microcontrollers. The primary reason for our choice is the simplicity of Saber. It uses power-of-two modulus which avoids modular reduction and rejection sampling during the generation of the public matrix entirely. Using the power of 2 modulus precludes the use of the asymptotically faster number theoretic transform (NTT) based polynomial multiplication. But as the polynomials are of small dimensions, we show that this does not cause large degradation in performance. Moreover, Saber uses the 'Learning with rounding (LWR)' problem which halves the amount of randomness compared to the 'Learning with Errors (LWE)' based schemes (e.g., Kyber [11]) and additionally reduces the bandwidth of communication. Another attractive feature of the Saber scheme is the use of 'Module' lattices instead of more commonly used ideal or standard lattices. The usage of modules facilitates scaling up or down to a higher or lower security as required by the application with minimum change in the code. This feature is also very useful for IoT applications. Furthermore, optimized software implementations of these KEM schemes on high-end processors, such as Intel processors with SIMD support, have been proposed and have been shown to be very efficient. But, to the best of our knowledge there are no published implementations of post-quantum KEM schemes on resource-constrained platforms. Due to these features we chose of Saber KEM scheme to investigate its implementation aspects in resource-constrained microcontrollers. However, we would like to remark that other post-quantum schemes could also be interesting for design and analysis on resource-constrained microcontrollers.

Our Contributions: In this paper, we present an efficient implementation of the module-LWR based KEM scheme Saber on the ARM Cortex-M0 and Cortex-M4 microcontrollers[1]. We achieve high speed on Cortex-M4 and a record minimum in the stack memory consumption in Cortex-M0. Our contributions are as follows:

1. We provide a full CCA-secure implementation of KEM operations namely keypair generation, encapsulation, and decapsulation on these microcontrollers. To the best of our knowledge these are the first published implementations of a CCA-secure post-quantum KEM on such resource-constrained microcontrollers. Our implementation on Cortex-M0 targets low memory footprint, and our implementation on Cortex-M4 aims for high-speed. Moreover, our design is very modular in nature *i.e* the optimization methods can be swapped with each other with little code rewrite to adapt

[1]The source codes are available in https://github.com/KULeuven-COSIC/TCHES2018_SABER

the implementations for a different microcontroller according to its computational capabilities or use case.

2. Due to the use of module-lattice, Saber consumes more memory in comparison to ideal-lattice based schemes. For e.g., the public matrix of Saber contains nine polynomials each having 256 coefficients, a total of 3.66 KB. Whereas in ideal-lattice based schemes with similar level of security only one polynomial with 768 coefficients need to be stored. In this work, we exploit the modular structure of the public matrix in Saber and propose a just-in-time strategy to perform the computations involving the public matrix without storing the matrix fully in the memory. We need only 0.5 KB of memory instead of 3.66 KB using this strategy. Furthermore, we reduce memory requirement during the generation of the secret vector.

   At the low-level, we minimize memory requirement for polynomial multiplications. The public domain implementation of Saber [17] uses the Toom-Cook method [29] combined with the Karatsuba method [27] for polynomial multiplication. These two algorithms are recursive and not in-place, resulting in large memory footprint. We use a less known in-place version of the Karatsuba multiplication algorithm [38] that only requires extra space for $n/4$ additional coefficients, where n is the number of coefficients in the polynomials. Though, the algorithm is slower than the Toom-Cook or the normal Karatsuba multiplication, we show that due to our optimization the performance remains within the admissible practical range for low-power processors. Our most memory efficient implementation in Cortex-M0 takes $4,786$K, $6,328$K, and $7,509$K clock cycles for key pair generation, encapsulation, and decapsulation respectively using at most 6.2 KB of memory.

3. Polynomial multiplication based on the NTT is believed to be the most efficient multiplication method due to its $O(n \log n)$ complexity and hence, parameters in most lattice-based schemes are chosen so that NTT can be used. However, in this paper we prove that a combination of Toom-Cook, Karatsuba and low-degree schoolbook multiplications can outperform optimized NTT-based polynomial multiplications. Since a polynomial is stored in a sequential addressable memory which is an SRAM in our platform, the overhead of memory access plays a critical role in the overall performance and is not captured in the asymptotic notion of the algorithmic complexity. We propose optimization techniques that carefully use the ALU-registers to accommodate several coefficients of the polynomials at once and then compute as much as possible involving the *local* coefficients, thus effectively reducing the number of sequential memory accesses.

   Furthermore, to achieve high speed, we effectively use DSP (digital signal processing) instructions of the Cortex-M4 platform to optimize the most frequently used schoolbook multiplication routine. This is possible due to the parameter choices of Saber. By utilizing these DSP instructions we reduce the number of multiplication instructions by approximately 34% in each execution of the schoolbook multiplication routine. Our polynomial multiplication competes with state-of-the-art NTT-based polynomial multiplications [6] and is even faster than the assembly-optimized implementation in [18, 33] by 1.6 and 3.4 times respectively for the same polynomial degree and on the same platform. Our fastest implementation in Cortex-M4 takes $1,147$K, $1,444$K, and $1,543$K clock cycles for key pair generation, encapsulation, and decapsulation respectively.

Furthermore, we stress that as the reference implementation of Saber our implementation does not have any data dependent branches and always runs on constant-time.

Organization of the paper: In Sec. 1 we provide a brief background on the lattice problems, the Saber KEM scheme, and methods for efficient polynomial multiplication.

In Sec. 2 we show how we optimized the building blocks used in Saber for achieving fast computation and reducing memory footprint. Experimental results using different combinations of these optimizations on Cortex-M0 and Cortex-M4 are provided in the next section. The final section draws the conclusions.

1 Background

In this section, we present a brief overview of the module-LWR problem, the Saber scheme that we will be optimizing, the platforms used in our implementation and the other cryptographic primitives that we will be using in our implementation.

1.1 LWE, LWR and module-LWR problem

The 'Learning With Errors' (LWE) problem, introduced by Regev [36], serves as a foundation of many lattice-based cryptographic schemes. The decisional LWE problem states that it is extremely hard to distinguish with non-negligible advantage between uniformly random samples from $\mathcal{U}(\mathbb{Z}_q^{l\times 1} \times \mathbb{Z}_q)$ and the same number of samples $(\boldsymbol{a}, b)$ of the form

$$\left(\boldsymbol{a}, b = \boldsymbol{a}^T \boldsymbol{s} + e\right) \in \mathbb{Z}_q^{l\times 1} \times \mathbb{Z}_q \,, \tag{1}$$

where $\boldsymbol{s} \in \mathbb{Z}_q^{l\times 1}$ is a fixed secret vector, $\boldsymbol{a} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l\times 1})$ are fresh uniformly random vectors and $e \leftarrow \beta_\mu(\mathbb{Z}_q)$ are fresh and small error terms sampled from an appropriate error distribution.

The 'Learning With Rounding' (LWR) problem was first introduced by Banerjee et al. [9] to create pseudo-random functions using lattices. Unlike the LWE problem, where the small error terms e are sampled from an error distribution, the error terms in LWR come from the rounding errors that are introduced inherently by scaling the samples from $\mathbb{Z}_q$ to $\mathbb{Z}_p$ where $p < q$. A decision-LWR problem is distinguishing samples of the following form,

$$\left(\boldsymbol{a}, b = \left\lfloor \frac{p}{q}(\boldsymbol{a}^T \boldsymbol{s}) \right\rceil\right) \in \mathbb{Z}_q^{l\times 1} \times \mathbb{Z}_p \tag{2}$$

from the same number of samples drawn from $\mathcal{U}(\mathbb{Z}_q^{l\times 1} \times \mathbb{Z}_p)$ for a fixed secret vector $\boldsymbol{s} \leftarrow \beta_\mu(\mathbb{Z}_q^{l\times 1})$ and fresh uniform random vectors $\boldsymbol{a}$. The computational LWR problem asks to recover the secret $\boldsymbol{s}$ from the LWR samples in Eq. 2. Reduction from LWE to LWR by Banerjee et al. [9] required q to be exponential in p, which was not suitable for cryptographic protocol constructions, but later works by Alwen et al. [8], Bogdanov et al. [10] and, Alperin-Sheriff et al. [7] required q to be polynomial in p, thus making the LWR problem convenient for construction of cryptographic schemes.

However both LWE and LWR-based constructions are inefficient due to matrix-vector arithmetic and require large memory. Following Lyubashevsky et al [32], faster computation and compact key can be achieved by using the ring-LWE and ring-LWR problems that use ideal lattices. There is plenty of theoretical research that have used the ring-LWE problem to construct a wide range of schemes such as public-key encryption [31, 32], key exchange [5, 13, 12], digital signature [19, 4] and homomorphic encryption [22, 15, 14]; as well as practical research that have implemented these schemes in hardware and software [39, 34, 25, 37]. However, the improvements come at a cost that the corresponding worst-case problem is now restricted to ideal lattices only, making the ring-LWE or ring-LWR-based schemes potentially susceptible to more avenues of attacks in the future.

Module-LWE [30] interpolates between standard lattice-based LWE and ideal lattice-based ring-LWE; elements are now polynomials in $R_q = \mathbb{Z}_q/(x^n + 1)$ unlike LWE where the elements are integers, and there is more than one polynomial unlike ring-LWE where

Algorithm 1: `Saber.KeyGen()` [17]

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

Algorithm 2: `Saber.Enc`$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in \mathcal{M}; r)$ [17]

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

only one polynomial is used. The security of a module-LWE instance is determined by the dimension l of the vectors and the degree of the polynomials. Thus, module-LWE offers a flexible combination of LWE and ring-LWE. Recently, the module-LWE problem has been used to design the CCA-secure post-quantum key-encapsulation mechanism Kyber [11] and the signature scheme Dilithium [20]. The security of Saber relies on the module-LWR problem, which is similar to the module-LWE problem but uses rounding to introduce the errors.

1.2 The Saber KEM

The IND-CPA key generation, encryption and decryption algorithms used in Saber [17] are shown in Alg. 1, 2 and 3. During key generation, a 32 byte random seed $seed_{\boldsymbol{A}}$ is generated, then it is expanded using the extendable output function `SHAKE-128` to construct the pseudorandom public matrix $\boldsymbol{A}$ of dimension $l \times l$. The secret $\boldsymbol{s}$ is a vector of dimension l and is sampled from a centered binomial distribution β_μ with parameter μ. The vector $\boldsymbol{b}$ is computed by performing a matrix-vector multiplication of $\boldsymbol{A}$ and $\boldsymbol{s}$ followed by an addition of a constant vector $\boldsymbol{h}$ and then bit-selection (using the `bits`) from the result. The public-key consists of $\boldsymbol{b}$ and the seed. The encryption operation uses matrix generation, binomial sampling, matrix-vector multiplication and bit selection. The decryption operation is rather simple and uses vector-vector multiplication and bit selection.

A IND-CCA KEM scheme [17] is achieved by applying a post-quantum variant of the Fujisaki-Okamoto transform [24] that comes with post-quantum reduction from Jiang et al. [26]. In App. A we provide the encapsulation and decapsulation algorithms used in the CCA-secure Saber KEM. The readers may follow the original paper [17] for detailed description and provable security of these algorithms.

Parameter set: For around 180-bit of quantum-security, Saber uses matrix or vector dimension $l = 3$ and ring-dimension $n = 256$. The two moduli p and q are 2^{10} and 2^{13} respectively. The parameter of the binomial error distribution is $\mu = 8$.

Algorithm 3: $\texttt{Saber.Dec}(sk = \boldsymbol{s}, c_m, \boldsymbol{b'})$ [17]

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

1.3 Polynomial multiplication

Saber performs polynomial arithmetic in the ring $R_q = \mathbb{Z}_q[x]/\langle x^{256} + 1\rangle$. Polynomial multiplication is very costly and needs special care. Since q is not a prime number in Saber, we cannot use NTT-based polynomial multiplication as done in [33, 18, 2, 6, 39, 34]. Instead, we use a combination of Toom-Cook and Karatsuba polynomial multiplication algorithms. The algorithms are described as follows.

1.3.1 Karatsuba polynomial multiplication

The Karatsuba multiplication algorithm [27] uses a divide-and-conquer approach to achieve $O(n^{\log_2 3})$ time complexity. The input polynomials $f(x)$ and $g(x)$ are split into half-sized polynomials as $f(x) = f_0 + f_1 \cdot x^{n/2}$ and $g(x) = g_0 + g_1 \cdot x^{n/2}$ and then the product is computed as $f(x) * g(x) = f_0 \cdot g_0 + \big((f_0 + f_1) \cdot (g_0 + g_1) - f_0 \cdot g_0 - f_1 \cdot g_1\big) \cdot x^{n/2} + f_1 \cdot g_1 \cdot x^n$. The algorithm is applied recursively. We refer to this algorithm as the classical Karatsuba in the rest of this paper.

1.3.2 Memory-efficient Karatsuba polynomial multiplication

The classical Karatsuba algorithm requires an additional $O(n)$ memory at each recursive call. Thus it is rather "memory hungry" for multiplying large polynomials on resource constrained microcontrollers such as Cortex-M0. In 2009, Roche [38] proposed a modification of Karatsuba's algorithm that requires only $O(\log n)$ additional memory per recursion. For the input polynomials $f^{(0)}(x)$, $f^{(1)}(x)$ and $g(x)$ the algorithm computes $h(x) = h(x) + \big(f^{(0)}(x) + f^{(1)}(x)\big) \cdot g(x)$. It works by performing rearrangements of the output array between each of the three recursive calls so that all write back operations can be performed only on the output array of length $2n - 1$, as shown in Algorithm 4. The objective of these rearrangements is to prepare the arrays for the next recursive call as well as to cancel the terms that were added in the previous calls as a consequence of the multiply-and-accumulate nature of this algorithm.

1.3.3 Toom-Cook polynomial multiplication.

The Toom-Cook method [29] is a generalization of the Karatsuba method: each multiplicand polynomial is split into w polynomials each having n/w coefficients. This is known as w-way Toom-Cook multiplication. A Toom-Cook multiplication essentially consists of three main steps: splitting, evaluation, and interpolation. In [17], the authors of Saber used a four-way Toom-Cook multiplication (see Alg. 5) to split a 256 multiplication into seven 64×64 multiplications. In the splitting stage, each multiplicand polynomial A and B is split into four equal polynomials each having 64 coefficients as $A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0 \cdot y$, (and similarly for B) where $y = x^{64}$. In the evaluation phase, these polynomials are evaluated at the points $\{0, \pm 1, \pm\frac{1}{2}, 2, \infty\}$ resulting in weighted sums of polynomials A_i's and B_i's. These weighted sums are then multiplied with each other to create polynomials w_1 to w_7 (steps $3 - 9$ in Alg. 5). In the interpolation stage (steps $10 - 24$) these values are further processed to produce the result C.

Algorithm 4: Memory efficient Karatsuba `kara_mem` [38]

Input: Three polynomials $f^{(0)}(x)$, $f^{(1)}(x)$ and $g(x)$ and their degree n
Output: $h(x) = h(x) + \left(f^{(0)}(x) + f^{(1)}(x)\right) \cdot g(x)$ of degree $2n-1$

$k = n/2$
$h[k : 2k-1] = h[0 : k-1] + h[k : 2k-1]$
$h[3k-1 : 4k-2] = f^{(0)}[0 : k-1] + f^{(1)}[0 : k-1] + f^{(0)}[k : 2k-1] + f^{(1)}[k : 2k-1]$
recursive call: $h[k :]$ =`kara_mem`$(g[0 :], g[k :], h[3k-1 :], k)$;
$h[3k-1 : 4k-2] = h[k : 2k-1] + h[2k : 3k-2]$
recursive call: $h[0 :]$ =`kara_mem`$(f^{(0)}[0 :], f^{(1)}[0 :], g[0 :], k)$;
$h[2k : 2k-2] = h[2k : 3k-2] - h[k : 2k-2]$
$h[k : 2k-1] = h[3k-1 : 4k-2] - h[0 : k-1]$
recursive call: $h[2k :]$ =`kara_mem`$(f^{(0)}[k :], f^{(1)}[k :], g[k :], k)$;
$h[k : 2k-1] = h[k : 2k-1] - h[2k : 3k-1]$
$h[2k : 3k-2] = h[2k : 3k-2] - h[3k : 4k-2]$
return $h(x)$;

1.4 Platforms

The ARM Cortex-M family offers a wide range of microcontroller-oriented processors suitable for single chip embbedded applications (MCU), Application Specific Standard Product (ASSP) and System-on-Chip (SoC) applications. Their lowest cost processor is the ARM Cortex-M0 which is designed for low power and has a reduced instruction set architecture (ISA), the ARMv6-M. General data processing instructions only operate on 8 registers and most commercial instantiations of this processors are produced with either 8KB or 16KB of RAM, so it is a very resource constrained device. Therefore, we target this microcontroller for our low memory implementation of Saber. A more powerful microcontroller of the same family is the ARM Cortex-M4, which is designed for Digital Signal Control (DSC) and has the ARMv7-M ISA. This processor includes a Digital Signal Processing (DSP) unit that gives support for different variants of multiply-accumulate instructions to run in a single clock cycle as well as SIMD capabilities for data sizes smaller than the word size. Therefore, we target this microcontroller with our high speed implementation of Saber.

1.5 `SHAKE-128` extended output function

`SHAKE-128` is an extended output function standardized in FIPS 202 [3]. It follows a sponge construction. It uses `keccak_absorb()` to absorb the input into its internal state. Then it uses `keccak_squeezeblocks()` on this state repeatedly to provide pseudorandom output of arbitrary length. The `keccak_squeezeblocks()` returns pseudorandom bytes in the multiples of 168 bytes.

2 Implementation

In this section we describe how we have implemented Saber in the resource-constrained microcontrollers Cortex-M0 and M4. As described in Alg. 1, 2 and 3, the building blocks that we need are as follows:

1. The extendable output function `SHAKE-128` for generating pseudorandom bytes.
2. A sampler to generate the secret vector $\boldsymbol{s}$ from a centered binomial distribution β_μ, where the parameter $\mu = 8$.

Algorithm 5: Toom-Cook Algorithm [17]

Input: Two polynomials $A(x)$ and $B(x)$ of degree $n = 256$
Output: $C(x) = A(x) * b(x)$
// Splitting $A(x)$ into four polynomials of size 64
$A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{64}$
// Splitting $B(x)$ into four polynomials of size 64
$B(y) = B_3 \cdot y^3 + B_2 \cdot y^2 + B_1 \cdot y + B_0$
// 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
$w_2 = w_2 + w_5$
$w_6 = w_6 - w_5$
$w_4 = (w_4 - w_3)/2$
$w_5 = w_5 - w_1 - 64 \cdot w_7$
$w_3 = w_3 + w_4$
$w_5 = 2 \cdot w_5 + w_6$
$w_2 = w_2 - 65 \cdot w_3$
$w_3 = w_3 - w_7 - w_1$
$w_2 = w_2 + 45 \cdot w_3$
$w_5 = (w_5 - 8 \cdot w_3)/24$
$w_6 = w_6 + w_2$
$w_2 = (w_2 + 16 \cdot w_4)/18$
$w_3 = w_3 - w_5$
$w_4 = -(w_4 + w_2)$
$w_6 = (30 \cdot w_2 - w_6)/60$
$w_2 = w_2 - w_6$
return $C(y) = w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7$;

3. A routine to generate the public matrix $\boldsymbol{A}$ or its transpose in Alg. 1 and 2.
4. Routines for computing polynomial arithmetic namely addition and multiplication. These routines could be used to realize matrix-vector or vector-vector arithmetic calculations.
5. Bit-level manipulation routines for reconciliation-data generation during encryption and for reconciliation during decryption
6. For the final IND-CCA Saber KEM, we need SHA3-256 and SHA3-512.

For implementing the symmetric primitives, namely SHAKE-128, SHA3-256 and SHA3-512, we used the Keccak code package by Bertoni et al. [23]. The bit-level manipulation routines that are used in Saber during rounding, reconciliation data generation and reconciliation are cheap and have little computation overhead. For the remaining building blocks, we perform two categories of optimizations by taking care of the constraints of the underlying platforms: for Cortex-M0 we optimize the memory footprint and for Cortex-M4 we optimize the number of computation cycles. In the following sub-sections, we describe our optimization strategies.

2.1 Speed optimization

Polynomial arithmetic is a computationally critical part in Saber. The polynomials are stored in the memory of the microcontroller as arrays of 16-bit half-words, each half-word containing a coefficient of the polynomial. For fast computation of polynomial multiplication, we first apply the Toom-Cook method to split a 256-coefficient polynomial multiplication into seven 64×64 polynomial multiplications. Then each such polynomial multiplication is performed using the Karatsuba method [27] until the number of coefficients in the operand polynomials becomes 16. Each 16-coefficient polynomial multiplication is computed using the quadratic-complexity schoolbook method. Since the coefficients are stored in the sequential addressable memory which is SRAM in our implementation, the overhead of memory accesses plays a critical role in the overall performance, and is not counted in the notion of asymptotic complexity of the polynomial multiplication algorithms. In this paper, we show how we minimize the memory access overhead by designing efficient algorithms.

2.1.1 Memory access optimization in Toom-Cook evaluation and interpolation

In the evaluation phase of the four-way Toom-Cook algorithm (Alg. 5), weighted sums of the polynomials A_0 to A_3 and B_0 to B_3 are computed to construct the multiplicands of the next-level polynomial multiplications. If the evaluations are computed *horizontally*, i.e., compute the polynomials $A(y)$ and $B(y)$ for the particular evaluation point and then compute $A(y) * B(y)$, then the cost of memory access will be huge. This is because for each weighted sum we have to read all the coefficients of A_0 to A_3 and B_0 to B_3 from the memory.

Algorithm 6: Toom-Cook 4-way evaluation

Input: Two polynomials $A(x)$ and $B(x)$of degree $n = 256$
Output: Evaluation polynomials w_1 to w_7 as in Alg. 5

for $j = 0$ *to* 63 **do**
- $r_0 = A_0[j]$;
- $r_1 = A_1[j]$;
- $r_2 = A_2[j]$;
- $r_3 = A_3[j]$;
- $r_4 = r_0 + r_2$;// $A_0 + A_2$
- $r_5 = r_1 + r_3$;// $A_1 + A_3$
- $r_6 = r_4 + r_5$; $r_7 = r_4 - r_5$;
- $aws_3[j] = r_6$;
- $aws_4[j] = r_7$;
- $r_4 = 2*(r_0*4+r_2)$;// $8*A_0+2*A_2$
- $r_5 = r_1 * 4 + r_3$;// $4 * A_1 + A_3$
- $r_6 = r_4 + r_5$; $r_7 = r_4 - r_5$;
- $aws_5[j] = r_6$;
- $aws_6[j] = r_7$;
- $r_4 = 8 * r_3 + 4 * r_2 + 2 * r_1 + r_0$;
- $aws_2[j] = r_4$; $aws_7[j] = r_0$;
- $aws_1[j] = r_3$;

Repeat the above steps to generate weighted sums bws_1 to bws_7;
for $i = 1$ *to* 7 **do**
- $w_i = aws_i * bws_i$;

return w_1 *to* w_7;

Algorithm 7: Toom-Cook 4-way interpolation

Input: Evaluation polynomials w_1 to w_7 as in Alg. 5
Output: $C(x) = A(x) * B(x)$ as in Alg. 5

$C \leftarrow 0$;
for $j = 0$ *to* 126 **do**
- $r_1 = w_2[i]$; $r_4 = w_5[i]$;
- $r_5 = w_6[i]$; $r_0 = w_1[i]$;
- $r_2 = w_3[i]$; $r_3 = w_4[i]$;
- $r_6 = w_7[i]$;
- $r_1 = r_1 + r_4$; $r_5 = r_5 - r_4$
- $r_3 = (r_3 - r_2)/2$; $r_4 = r_4 - r_0$
- $r_8 = 64 \cdot r_6$; $r_4 = r_4 - r_8$
- $r_4 = 2 \cdot r_4 + r_5$; $r_2 = r_2 + r_3$
- $r_1 = r_1 - 65 \cdot r_2$; $r_2 = r_2 - r_6$
- $r_2 = r_2 - r_0$; $r_1 = r_1 + 45 \cdot r_2$
- $r_4 = (r_4 - 8 \cdot r_2)/_{24}$; $r_5 = r_5 + r_1$
- $r_1 = (r_1 + 16 \cdot r_3)/_{18}$; $r_3 = -(r_3 + r_1)$
- $r_5 = (30 \cdot r_1 - r_5)/_{60}$; $r_2 = r_2 - r_4$
- $r_1 = r_1 - r_5$; $C[i] = (C[i] + r_6)$;
- $C[64 + i] = (C[64 + i] + r_5)$;
- $C[128 + i] = (C[128 + i] + r_4)$;
- $C[192 + i] = (C[192 + i] + r_3)$;
- $C[256 + i] = (C[256 + i] + r_2)$;
- $C[320 + i] = (C[320 + i] + r_1)$;
- $C[384 + i] = (C[384 + i] + r_0)$;

return C;

To minimize the number of memory accesses, we adapt a *vertical* coefficient scanning

method. The jth coefficients of the four polynomials A_0 to A_3 are read in a batch from the memory and loaded in the registers of the processor. Then these registers are used to compute the jth coefficients of all of the weighted sums as required in the lines 4 to 8 in Alg. 5. We can do this easily as there are 14 usable general purpose registers in Cortex-M4 and this approach needs only 8 registers as shown in Alg. 6. In the algorithm, r_* represent the general purpose registers and $aws_*[\,]$ represent the arrays of the weighted sums that are stored in the memory. In Alg. 6, special care has been taken to minimize the number of arithmetic operations during this process. We repeat the same procedure to compute the weighted-sum arrays from B_0 to B_3. Unlike the horizontal method, the vertical method accesses each of the 64 coefficients of A_0 to A_3 and B_0 to B_3 only once. The overhead of this approach is that it requires ten additional arrays each of length 64 to store the weighted sums.

During the interpolation phase of the Toom-Cook algorithm, we apply a similar vertical technique: we load the jth coefficients of all of w_1 to w_7 in the internal registers and perform the arithmetic operations on the registers. Having multiple consecutive loads also helps to decrease the latency since atomic load instructions take three clock cycles, whereas batch loading of three coefficients takes only four cycles thanks to the pipelined datapath of Cortex-M4. The optimized steps are shown in Alg. 7.

To validate that this algorithm is more efficient, we have implemented both algorithms on C prior to realize any assembly optimization on the Toom-Cook function. In this setting, the *horizontal* algorithm executes a 256 coefficient multiplication in $83,550$ clock cycles while the *vertical* algorithm only requires $78,633$ clock cycles for the same multiplication, using both algorithms the same combination of two level Karatsuba and schoolbook behind Toom-Cook. Then, we have also carried out assembly optimizations to achieve our cycle counts as shown in Table 4.

2.1.2 Speeding up School book multiplication using DSP instructions

As Saber uses LWR, it needs multiplication for two different rings. Hence, the coefficients of each polynomials can be of either 10 or 13 bits long. We pack each coefficient of a polynomial in a 16-bit half-word. Two coefficients are then loaded in a single 32-bit full-word of the processor. Let r^i be a full-word register. We represent the bottom half-word by r^i_0 and the top half-word by r^i_1. We use the DSP multiply-and-accumulate instruction `SMLA` available in Cortex-M4 for multiplying two half-words and accumulating the result. The flags B or T in the instruction are used for choosing the bottom or top half-word respectively.

$$\texttt{SMLA}^{B/T B/T}(r^a, r^b, r^c, r^d) \coloneqq r^a \leftarrow r^b_{0/1} * r^c_{0/1} \;+\; r^d$$

We use this instruction multiply two coefficients or to compute the halfword multiplications like a_0b_0, a_2b_0 as shown in Fig. 1.

Now, consider again the example shown in Fig. 1 where we multiply four coefficients of polynomial $A(x)$ and four coefficients of polynomial $B(x)$ and accumulate the result in the polynomial $C(x)$. In the naive way, we have to use 16 `SMLA` instructions in this computation. We use the DSP instruction `SMLADX` that facilitates cross multiplication between half-words of registers and perform two multiply-and-accumulate operations (marked in blue dashed rectangles) simultaneously.

$$\texttt{SMLADX}(r^a, r^b, r^c, r^d) \coloneqq r^a \leftarrow r^b_0 * r^c_1 \;+\; r^b_1 * r^c_0 \;+\; r^d$$

This effectively reduces the total number of multiplication instructions in Fig. 1 from 16 to 12. For multiplying two polynomials with 16 coefficients we need only 192 multiply-and-accumulate operations instead of 256. Further, with some arrangements of the coefficients we can do even better. In the beginning of each inner loop of a multiplication, we pack two adjacent coefficients that are in different registers (e.g., coefficients b_1 and b_2) in a

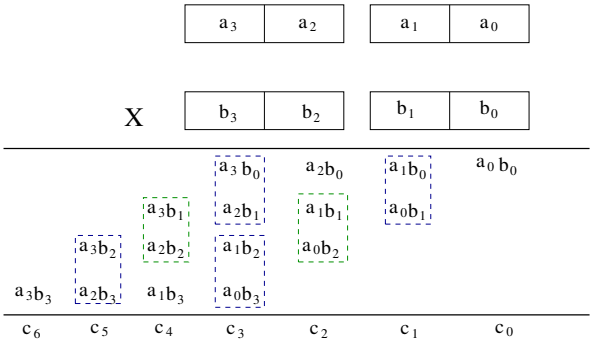

Figure 1: Reducing the number of multiply-and-accumulate instructions in schoolbook multiplication.

spare register using `PKHBT` instruction and then perform cross multiplication using `SMLADX` instruction as explained above(marked in green dashed rectangles). In our assembly routine for schoolbook multiplication, we are able to fit a maximum of eight coefficients of one multiplicand polynomial and four coefficients of the other multiplicand polynomial at a time due to our optimized usage of internal registers. As the input polynomials to our schoolbook multiplication routine are always of 16 coefficients, the inner loop of our schoolbook multiplication runs only eight times. Thus in each inner loop we can save four instructions due to the rearrangement of coefficients at the cost of one extra `PKHBT` for the rearrangement. Hence, we can save three instructions per iteration of the inner loop. So, ultimately we need only 168 multiply-and-accumulate instructions instead of 256 instructions resulting in approximately 34% less multiply-and-accumulate instructions. Our assembly-optimized schoolbook multiplication takes only 587 clock cycles for a 16×16 polynomial multiplication. It should be noted that this method can not be used in microcontrollers without similar DSP instructions, for example Cortex-M0.

2.2 Memory optimization

The reference C implementation of Saber submitted for NIST PQC standardization process [1] consumes 16.3 KB of stack memory when compiled for the Cortex-M4 platform. In this section, we describe the optimization tricks that we applied in our implementation to reduce the memory footprint to only 6.2 KB at most.

2.2.1 Generation of the public matrix $\boldsymbol{A}$

In Saber, the matrix is composed of nine polynomials, each having 256 coefficients of 13 bits each. In the reference implementation of Saber [17], `SHAKE-128` is used to generate a total of $9 \cdot 256 \cdot 13/8 = 3,744$ pseudorandom bytes in a byte-bank. Next, the polynomials of $\boldsymbol{A}$ are generated in the row-major order by packing the bytes into 13-bit coefficients. In this way, the generation of $\boldsymbol{A}$ during the key-pair generation, or the generation of $\boldsymbol{A}^T$ during the encryption operation requires roughly 3.8 KB of memory. We reduce this memory requirement using a just-in-time approach which is explained as follows.

In Alg. 1, following the generation of $\boldsymbol{A}$, a matrix-vector multiplication with $\boldsymbol{s}$ is performed and the result is stored in the vector $\boldsymbol{b}$. In a matrix-vector multiplication, an element which in this case is a polynomial in R_q from the matrix, is used only once. Hence, we generate the required polynomial just-in-time, use it in the polynomial multiplication, and then reuse the space occupied by it for the next polynomial. Thus, the memory requirement reduces from 9 polynomials to just one polynomial. However, this requires a major book keeping in the `SHAKE-128` function.

The `keccak_squeezeblocks()` function inside `SHAKE-128` outputs 168 bytes at a time. When the goal is to minimize the memory requirement, the best approach would be to call `keccak_squeezeblocks()` once, then pack the output bytes into the coefficients of the target polynomial, and repeat this process until the target polynomial is constructed. But this approach results in two types of leftovers.

To construct a polynomial in R_q, we require 468 bytes, which is not a multiple of 168. Hence there are bytes that are leftover after constructing a polynomial. The second leftover is at the bit-level. From 168 bytes, we can construct 103 coefficients and thus there are $168 \cdot 8 - 103 \cdot 13 = 5$ bits of leftover. Now these leftover bits need to be placed in the start position of the byte-bank before storing the next 168 bytes from the next `keccak_squeezeblocks()` call. This incurs costly bit-level manipulation. Another problem is that the number of leftover bits and bytes change in every step. In Table 1, we have shown the number of leftover bits and bytes during the construction of the first polynomial of $\boldsymbol{A}$. Such an implementation requires major book-keeping and a fully unrolled implementation of it would significantly increase the code size.

Steps	**Polynomial coefficients**	**Leftover bits**	**Leftover bytes**
$\texttt{Squeezeblocks}_0\texttt{()} \rightarrow$ 168 bytes	103	5	0
$\texttt{Squeezeblocks}_1\texttt{()} \rightarrow$ 168 bytes	103	2	1
$\texttt{Squeezeblocks}_2\texttt{()} \rightarrow$ 168 bytes	50	0	88

Table 1: Leftover bits and bytes during the construction of the first polynomial of $\boldsymbol{A}$

To avoid bit-level manipulation, we pack the maximum number of available bytes, say m, that is a multiple of 13. Thus the packing outputs $m \cdot 8/13$ coefficients of the target polynomial. The leftover bytes are then moved to the initial portion of the byte-bank. Next, `keccak_squeezeblocks()` is called again and the output of it is written in the byte-bank just after the leftover bytes. Now, the bytes in the byte-bank are packed into the next coefficients of the target polynomial following the same way. The steps are explained in Fig. 2. Following this approach, the size of the byte-bank is set to 280 bytes, since the number of leftover bytes during the construction of $\boldsymbol{A}$ could be at most 112.

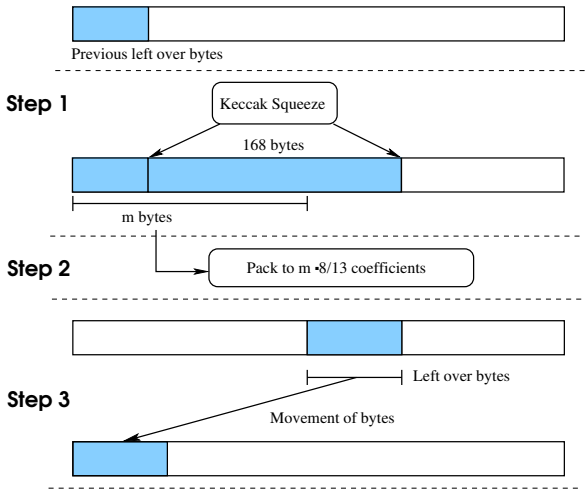

Figure 2: Use of byte-bank during generation of polynomials of $\boldsymbol{A}$. Turquoise blue color is used to indicate the bytes that will be used.

2.2.2 Secret generation

The reference implementation of Saber uses a byte-bank of length 768 to store all the required pseudorandom bytes obtained from `SHAKE-128` for generating the three polynomials of the secret vector $\boldsymbol{s}$. To save memory, we use a just-in-time method similar to the method used in the generation of the public matrix $\boldsymbol{A}$. We call `keccak_squeezeblocks()` to produce 168 bytes at a time in the byte-bank, and then we use the pseudorandom bytes for constructing the binomially distributed coefficients of the target secret vector. Similar to the generation of the public matrix, we have leftover bytes since the output length of `keccak_squeezeblocks()` is not a multiple of 256. We post-process the leftover bytes in a similar way that we followed during the generation of the public matrix. Since there can be at most 80 bytes of leftover, the length of the byte-bank is only 248. Thus we save approximately 0.5 KB of memory w.r.t the reference implementation.

Beside these two major optimizations, we performed optimizations in the high-level source code to reuse memory as much as possible. These small scale optimizations contributed in maximum reduction of the overall stack memory.

2.2.3 Memory efficient Karatsuba

The memory efficient version of Karatsuba introduced in Section 1.3.2 has two disadvantages. First, the performance suffers a penalization in terms of latency with respect to the classical Karatsuba because of the continuous arrangements between multiplications. To illustrate this, a plain C version of that algorithm performs a 64 coefficient multiplication within $17,743$ clock cycles while a multiplication of the same magnitude only takes $12,482$ when utilizing a version of the classical Karatsuba algorithm. Second, the reduction from $O(n)$ to $O(\log n)$ in the extra memory requirements might not be enough to make the algorithm suitable for low memory platforms. To mitigate these disadvantages, we implement an ad hoc version of this algorithm adapted for our parameters.

For the sake of modularity, we decided to keep the depth of Karatsuba's recursion a multiple of two, so we unrolled two levels of iteration of Karatsuba into one. This routine consists of nine calls to a multiplication routine with polynomials one fourth of size of its original input polynomials and some corresponding arrangements before and after each of these calls. This is shown in Alg. 8. In each of these arrangements, we can combine and reorganize the array scans to reduce the number of memory accesses by adopting the vertical coefficient scanning method as used in the Toom-Cook algorithm described in Sec. 2.1.1. Moreover, some of these arrangements between the original recursive calls in Alg. 4 can be merged and this also leads to eliminate certain redundancy on these arrangements.

This is illustrated in Alg. 9, which represents `arrangements_7()` as an example. These arrangements lie between the end of the second recursive call and the beginning of the third recursive call according to the original algorithm in Alg. 4. If we consider a two-level Karatsuba then this arrangement is equivalent to perform two subtractions over $k/2$ coefficients, followed by two subtractions over k coefficients, and five additions over $k/2$ coefficients. Without the unroll and merge approach, this translates into $22 \times k/2$ load operations and $11 \times k/2$ store operations. If we analyze Alg. 9, then the same can be achieved using only $14 \times k/2$ loads and $7 \times k/2$ stores, thus reducing memory access by 36.4%. A detailed description of the full algorithm can be found in App. C. As an experiment, we measured the clock cycles required by the actual memory efficient Karatsuba (Alg. 4) and our modified version of it on a Cortex-M4 for 64-coefficient polynomials and two levels of recursions. We found that the cycle counts decreased from $17,473$ to only $6,203$ excluding the cost of threshold-level schoolbook multiplications in both cases. This speedup is indeed remarkable considering the recursive nature of the Karatsuba algorithm where even a small reduction in every recursion would result in big reduction in the overall multiplication.

To reduce the memory requirements, we consider $f^{(0)}(x) = f(x)$ and $f^{(1)} = 0$ on the top level as well as in all recursive calls and perform the additions $f^{(0)}(x) + f^{(1)}(x)$ in-place before each call to a multiplication algorithm. Thus, our Karatsuba algorithm does not compute $h(x) = h(x) + (f^{(0)}(x) + f^{(1)}(x)) * g(x)$ but $h(x) = h(x) + f(x) * g(x)$ saving an entire polynomial of memory space. The extra memory required by Karatsuba, which is used for the in-place additions before each call to a multiplication routine, is now allocated on each recursive call. Considering that the function was called to perform a multiplication of n coefficients, the extra memory will be a space for $n/4$ coefficients This is enough because the recursive call will be applied over $n/4$. For our parameters, Algorithm 4 require 512 bytes of extra memory for storing the 256 coefficients of $f^{(1)}(x)$, while we only need 128 bytes to store 64 coefficients saving 384 bytes of stack. For our implementation on Cortex-M0 we use 4 levels of Karatsuba recursion *i.e* two of our 2-level unrolled Karatsuba. We found experimentally that this is the optimal choice for multiplying two 256 coefficient polynomials using Karatsuba algorithm.

Algorithm 8: High-level description of unrolled memory efficient Karatsuba

Input: Pointers to $f(x)$ and $g(x)$ and their degree n
Output: $h(x) = h(x) + f(x) * g(x)$ of degree $2n - 1$

1 extra memory $tmp[n/4]$
2 $k = n/2$
3 `arrangements_1`(f, g, h, k)
4 `poly_mul`$(tmp, h[2k + k/2], h[k + k/2], k/2)$
5 `arrangements_2`(f, g, h, k)
6 `poly_mul`$(tmp, h[3k - 1], h[k], k/2)$
7 `arrangements_3`(f, g, h, k)
8 `poly_mul`$(tmp, h[3k + k/2 - 1], h[2k], k/2)$
9 `arrangements_4`(f, g, h, k)
10 `poly_mul`$(tmp, h[k + k/2 - 1], h[k/2], k/2)$
11 `arrangements_5`(f, g, h, k)
12 `poly_mul`$(f, g, h, k/2)$
13 `arrangements_6`(f, g, h, k)
14 `poly_mul`$(f[k/2], g[k/2], h[k], k/2)$
15 `arrangements_7`(f, g, h, k)
16 `poly_mul`$(tmp, h[3k + k/2 - 1], h[2k], k/2)$
17 `arrangements_8`(f, g, h, k)
18 `poly_mul`$(f[k], g[k], h[2k], k/2)$
19 `arrangements_9`(f, g, h, k)
20 `poly_mul`$(f[k + k/2], g[k + k/2], h[2k], k/2)$
21 `arrangements_10`(f, g, h, k)
22 **return** $h[x]$;
23 $^{(*)}$ skip last iteration

Algorithm 9: Example:`arrangements_7`

Input: Pointers to $f(x)$, $g(x)$, $h(x)$ and value of k
Output: Modifies parts of $h(x)$ and tmp

1 **for** $i = 0$ *to* $k/2$ **do**
2 $h[k/2+i] = h[k/2+i] - h[k+i]$;
3 $h[2k + i] = h[2k + i] - h[k + i] + h[k + k/2 + i]^{[*]}$;
4 $h[k + i] = h[3k - 1 + i] - h[i]$;
5 $h[2k + k/2 + i] = h[2k + i] + h[2k+k/2+i] - h[k+k/2+i]^{[*]}$;
6 $h[k + k/2 + i] = h[3k+k/2-1+i] - h[k/2+i]$;
7 $h[3k + k/2 - 1 + i] = f[k + i] + f[k + k/2 + i]$;
8 $tmp[i] = g[k+i]+g[k+k/2+i]$;

2.2.4 High-level optimization: Saber#

As specified in Saber [17], the public matrix $\boldsymbol{A}$ is generated in row-major order during key generation Alg. 1 and in column-major order (i.e., transpose) during encryption Alg. 2. In the previous section, we showed that using a just-in-time strategy we can reduce the memory footprint of the public matrix. Since the matrix is generated in row-major order during the matrix-vector multiplication $\boldsymbol{As}$, we get the polynomials of the result vector

one by one. With this we need to store just one result-polynomial since the polynomial can be immediately rounded and packed into the target public-key. However, during the encryption operation we need to assign memory for all three result-polynomials of the matrix-vector multiplication $\boldsymbol{A}^T\boldsymbol{s}'$ since the matrix is generated in column-major order. The two situations are explained in Fig. 3. As encryption is the most complex operation and memory usage touches the peak, we could lower the peak by generating the public matrix in row-major order during encryption and in column-major order during key generation. This modification of course changes the scheme, hence we call it Saber#.

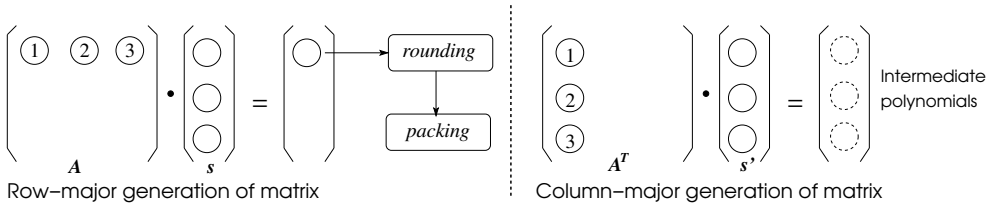

Figure 3: Matrix-vector multiplications during key generation and encryption. The numbers are used to indicate just-in-time generation of the polynomials of the matrices.

3 Results

We used the ARM-GCC toolchain to compile our source code with the flags `-O3` both for the Cortex-M0 the Cortex-M4. The clock-cycles are measured with inbuilt functions using a clock running at the same frequency as the processor. We executed the software $10,000$ times and found that each execution requires the same number of cycles thus supporting our claim that our implementation is indeed a constant-time implementation. We used a STM32F4-discovery board from STMicroelectronics to evaluate the performance on Cortex-M4 platform and a XMC2Go board from Infineon to evaluate the performance on Cortex-M0 platform. Both platforms are equipped with random number generators which we also use in our implementation to generate the seed bytes as specified in the Saber scheme.

As we mentioned earlier, our implementation is highly modular in nature. The optimizations described in Sec. 2.1 and Sec. 2.2 can be swapped easily with minimum amount of code rewriting. This provides great flexibility to trade-off between speed and memory usage. Therefore, in the first part of this section we show results on Cortex-M0 and Cortex-M4 with different combinations of our optimizations. In the later part of this section we will compare our implementation with other reported implementations. Fig. 4 and Fig. 5 show the variations in time and memory for different combinations of the proposed optimizations in Cortex-M0 and Cortex-M4 respectively. In the figures, the abbreviation *TC* refers to Toom-Cook, *kara_mem* to memory efficient Karatsuba and *kara_classic* to the classical Karatsuba algorithm, respectively.

Table 2 shows the time memory trade-off between Saber and Saber#. We can see that the optimization described in Sec. 2.2.4 decrease the memory consumption by more than one kilobytes in encapsulation and decapsulation.

	Cortex-M0 (4-kara_mem)			Cortex-M4 (TC + kara_mem + memory opts.)		
	Keypair	Encaps.	Decaps.	Keygen	Encaps.	Decaps.
Saber	$4,786/5,031$	$6,328/5,119$	$7,509/6,215$	$1,165/6,932$	$1,530/7,019$	$1,635/8,115$
Saber#	$4,782/5,007$	$6,325/4,079$	$7,507/5,175$	$1,162/6,923$	$1,528/5,987$	$1,633/7,163$

Table 2: Speed/memory requirements for Saber and Saber#. Table entries are in cycles$\times 10^3$/memory (bytes).

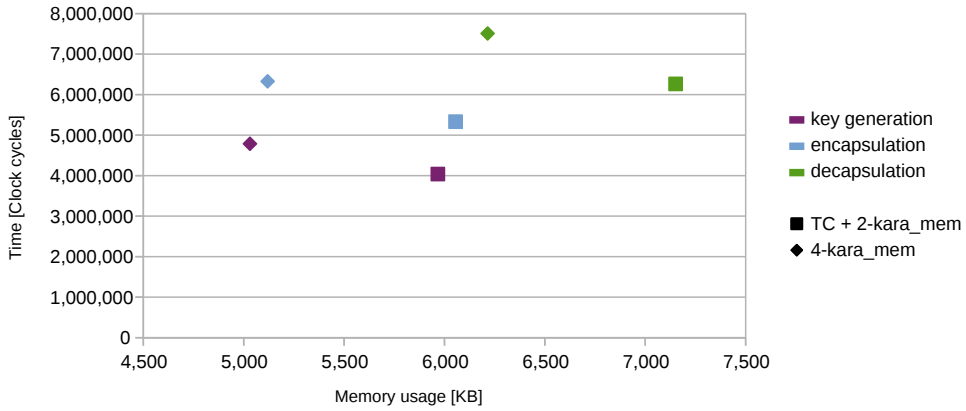

Figure 4: Time vs memory for different combinations of optimizations in Cortex-M0.

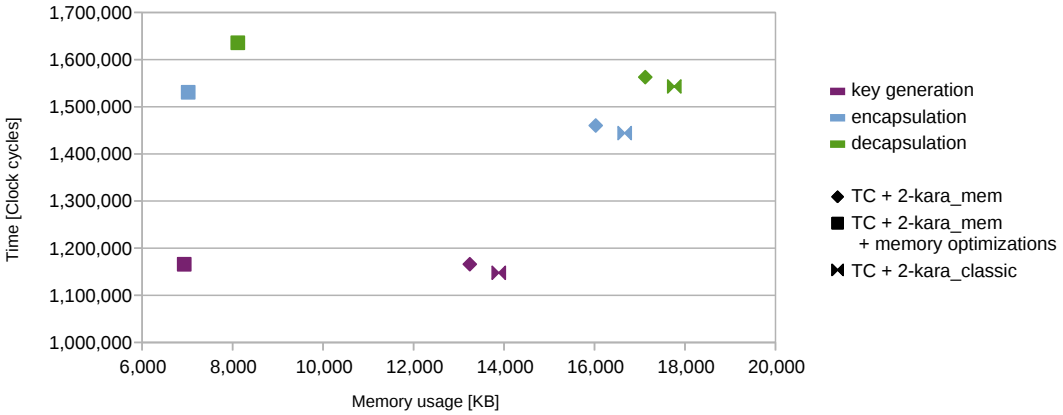

Figure 5: Time vs memory for different combinations of optimizations in Cortex-M4.

Comparisons

In Table 3, we compare our speed-optimized and memory-optimized implementations of Saber with other lattice based key exchange and encryption schemes in ARM Cortex-M series microcontrollers. Since NIST announced the list of submitted post-quantum cryptography proposals very recently, we could compare with only a limited number of optimized software implementations available in the concurrent work pqm4 project [2] on similar resource-constrained platforms for the schemes `NewHope-CCA`, `Frodo` and `Kyber`. From the table we see that the speed-optimized implementation of Saber is faster than `NewHope-CCA` and `Frodo` in all aspects. Saber is faster than Kyber in key generation and encapsulation, but marginally slower in decapsulation. In a computationally asymmetric (e.g. RFID tag and server) key establishment, the weaker party computes the encapsulation and the powerful server computes the key generation and decapsulation. Hence, faster encapsulation which happens in Saber, is beneficial for computationally asymmetric key establishment. As can be seen from the table, our memory-optimized implementation achieves a very low memory consumption with only 6.2 KB in the XMC2Go board which has Cortex-M0 processor and 16 KB of memory. Yet, the implementation takes at most 7509K clock cycles for the most expensive decapsulation operation. These results show that Saber is very efficient on resource-constrained platforms.

We remark that such comparisons with different cryptosystems are nearly impossible as they are very different from each other in many ways like underlying hard problems on

	Platform	Key generation	Encapsulation	Decapsulation
Frodo	Cortex-M4	94,191K cycles	111,688K cycles	112,156K cycles
		36,536 bytes	58,328 bytes	68,680 bytes
NewHope-CCA	Cortex-M4	1,246K cycles	1,966K cycles	1,977K cycles
		11,160 bytes	17,456 bytes	19,656 bytes
Kyber	Cortex-M4	1,200K cycles	1,497K cycles	1,526K cycles
		10304 bytes	13464 bytes	14624 bytes
Saber-speed	Cortex-M4	1,147K cycles	1,444K cycles	1,543K cycles
		13,883 bytes	16,667 bytes	17,763 bytes
Saber-memory	Cortex-M4	1,165K cycles	1,530K cycles	1,635K cycles
		6,931 bytes	7,019 bytes	8,115 bytes
Saber-memory	Cortex-M0	4,786K cycles	6,328K cycles	7,509K cycles
		5,031 bytes	5,119 bytes	6,215 bytes

Table 3: Comparisons with other reported implementations of lattice-based schemes in ARM Cortex-M microcontrollers in terms of of time and memory requirements.

which they are based, choice of parameters, levels of security etc. We should also mention that for our optimized version we only optimized the polynomial multiplication. The memory efficient optimization on Cortex-M0 does not have any assembly optimization so we think the speed will improve further upon careful optimization. But we leave that for future research.

Table 4 shows the number of clock cycles required to perform a single 256×256 polynomial multiplication. Since [33, 6] report results for NTT-based multiplications of 512 and 1024-coefficient polynomials, we scaled down the cycle counts by factor 2.25 and 5 respectively considering the asymptotic $O(n \log n)$ complexity of the NTT. Here we can see that our fastest multiplication outperforms the NTT-based multiplications in [33, 18, 2]. To the best of our knowledge we gain this advantage over NTT-based multiplications firstly because our modulus is a power-of-two and thus we do not spend any time for modular reduction. Secondly, NTT has a complex butterfly structure which requires access of non-consecutive memory words. Whereas, our combination of Toom-Cook, Karatsuba and schoolbook methods access memory in a consecutive fashion and hence we can use the DSP instructions in a better way than the NTT based multiplications. These expensive memory access and modular reduction operations counteract the asymptotical advantage of the NTT-based polynomial multiplication.

4 Conclusions

In this paper, we proposed a speed optimized and a memory optimized software implementations of the CCA-secure lattice-based key encapsulation scheme Saber targeting resource-constrained platforms Cortex-M0 and Cortex-M4. Our memory optimized implementation shows that lattice-based cryptography, which is known for large memory requirements and relatively larger key sizes, can be practical on extremely resource-constrained Cortex-M0 platforms typically having only 8 KB of memory. This implementation takes 7,509K clock cycles which is less than a quarter of a second assuming 32 MHz frequency for the most time consuming decapsulation operation. Our speed optimized implementation running in the Cortex-M4 just takes 1,543K clock cycles and thus spends only 9 miliseconds assuming 168 MHz frequency to compute a decapsulation operation.

We have also shown that with suitable choice of parameters, asymptotically slower Toom-Cook and Karatsuba multiplications can get very competitive and even sometime outperform NTT-based multiplications. We believe our results will empower cryptographers

Implementations	Polynomial multiplication
Cortex-M4F [33]†	226,055
Cortex-M4F [18]‡	108,147
Cortex-M4 [2, 11] ▷	≈73705
Cortex-M4 [6]*	≈54447
Our ⊕	65,459
Our ⋆	66,692

† Reported 508,624 cycles for polynomial degree 512 and prime modulus 12289.

‡ Reported 108,147 cycles for polynomial degree 256 and prime modulus 7681.

∗ Reported ≈ 272,235 cycles for polynomial degree 1024 and prime modulus 12289.

▷ Polynomial degree 256 and prime modulus 7681

⊕ Speed-optimized implementation. Toom-Cook+classical Karatsuba+schoolbook

⋆ Speed-optimized implementation. Toom-Cook+memory-efficient Karatsuba+schoolbook

Table 4: Comparison of clock cycles for 256×256 polynomial multiplications with scaling when necessary. The cycle counts for NTT based multiplications are calculated as cycle count for 2*Forward NTT + Inverse NTT.

to select parameters for their lattice-based schemes from a wider range of choices, which were earlier limited to prime moduli and power-of-two polynomial rings due to the restrictions posed by the NTT. We conclude with the hope that this paper will be useful for the ongoing NIST standardization process to select scheme(s) for the post-quantum world.

5 Acknowledgements

This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work was supported by the European Commission through the Horizon 2020 research and innovation programme under grant agreement Cathedral ERC Advanced Grant 695305 and by EU H2020 project FENTEC (Grant No. 780108) and by the Hercules Foundation AKUL/11/19.

References

[1] Nist post-quantum cryptography round 1 submissions. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions, 2017. [Online; accessed 12-April-2018].

[2] pqm4 post-quantum crypto library for the arm cortex-m4. https://github.com/mupq/pqm4, 2018. [Online; accessed 15-April-2018].

[3] National Institute of Standards and Technology. 2015. SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, 2015.

[4] S. Akleylek, N. Bindel, J. Buchmann, J. Krämer, and G. A. Marson. An efficient lattice-based signature scheme with provably secure instantiation. In D. Pointcheval, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2016: 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, pages 44–60. Springer International Publishing, Cham, 2016.

[5] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange – a new hope. In *USENIX Security 2016*, 2016.

[6] E. Alkim, P. Jakubeit, and P. Schwabe. Newhope on arm cortex-m. In C. Carlet, M. A. Hasan, and V. Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 332–349, Cham, 2016. Springer International Publishing.

[7] J. Alperin-Sheriff and D. Apon. Dimension-preserving reductions from lwe to lwr. Cryptology ePrint Archive, Report 2016/589, 2016.

[8] J. Alwen, S. Krenn, K. Pietrzak, and D. Wichs. Learning with rounding, revisited - new reduction, properties and applications. In *CRYPTO 2013*, pages 57–74, 2013.

[9] A. Banerjee, C. Peikert, and A. Rosen. *Pseudorandom Functions and Lattices*, pages 719–737. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] A. Bogdanov, S. Guo, D. Masny, S. Richelson, and A. Rosen. On the hardness of learning with rounding over small modulus. In *13th International Conference on Theory of Cryptography*, pages 209–224, 2016.

[11] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[12] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *CCS 2016*, pages 1006–1018. ACM, 2016.

[13] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, May 2015.

[14] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 868–886. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, Oct 2011.

[16] L. Chen, S. P. Jordan, Y.-K. Liu, D. Moody, R. C. Peralta, R. A. Perlner, and D. C. Smith-Tone. Report on post-quantum cryptography. 2016.

[17] J.-P. DÁnvers, A. Karmakar, S. S. Roy, and F. Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. Cryptology ePrint Archive, Report 2018/230, 2018. https://eprint.iacr.org/2018/230 to appear in Africacrypt 2018.

[18] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 339–344, San Jose, CA, USA, 2015. EDA Consortium.

[19] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle. Crystals – dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. https://eprint.iacr.org/2017/633.

[21] R. P. Feynman. Simulating physics with computers, 1981. https://people.eecs.berkeley.edu/~christos/classics/Feynman.pdf.

[22] C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[23] Guido Bertoni and Joan Daemen and Michaël Peeters and Gilles Van Assche and Ronny Van Keer. Keccak implementation overview. https://github.com/gvanas/KeccakCodePackage, 2012. [Online; accessed 12-April-2018].

[24] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the fujisaki-okamoto transformation. Cryptology ePrint Archive, Report 2017/604, 2017. http://eprint.iacr.org/2017/604.

[25] J. Howe, C. Rafferty, A. Khalid, and M. O'Neill. Compact and provably secure lattice-based signatures in hardware. In *IEEE International Symposium on Circuits and Systems, ISCAS 2017, Baltimore, MD, USA, May 28-31, 2017*, pages 1–4, 2017.

[26] H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma. Post-quantum ind-cca-secure kem without additional hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[27] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of USSR Academy of Sciences*, 145(7):293–294, 1962.

[28] J. Kelly. A preview of bristlecone, google's new quantum processor, 2018. [Online; accessed 10-April-2018].

[29] D. Knuth. *The Art of Computer Programming, Volume 2. Third Edition*. Addison-Wesley, 1997.

[30] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, Jun 2015.

[31] R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In A. Kiayias, editor, *Topics in Cryptology – CT-RSA 2011: The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, pages 319–339. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[32] V. Lyubashevsky, C. Peikert, and O. Regev. *On Ideal Lattices and Learning with Errors over Rings*, pages 1–23. Springer Berlin Heidelberg, 2010.

[33] T. Oder, T. Pöppelmann, and T. Güneysu. Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[34] T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 353–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[35] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *eprint arXiv:quant-ph/0301141*, Jan. 2003.

[36] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC '05*, pages 84–93. ACM, 2005.

[37] C. P. Renteria-Mejia and J. Velasco-Medina. High-throughput ring-lwe cryptoprocessors. *IEEE Trans. VLSI Syst.*, 25(8):2332–2345, 2017.

[38] D. S. Roche. Space- and time-efficient polynomial multiplication. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2009, Seoul, Republic of Korea, July 29-31, 2009, Proceedings*, pages 295–302, 2009.

[39] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*, pages 371–391, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[40] P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.

A CCA secure Saber KEM

Algorithm 10: Saber.Encaps($pk = (\boldsymbol{b}, seed_{\boldsymbol{A}})$) [17]

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

The encapsulation and decapsulation operations used in the Saber KEM are described in Alg. 10 and 11 respectively. Two hash functions $\mathcal{G}$ and $\mathcal{H}$, which are implemented using SHA3-512 and SHA3-256 respectively, are used in the CCA transformation along with the IND-CPA Saber.Enc and Saber.Dec functions.

Algorithm 11: Saber.Decaps($sk = (\boldsymbol{s}, z), pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), c$) [17]

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

B Figure explaining memory efficient Karatsuba algorithm

Karatsuba's algorithm splits each of the input polynomials into two and then allows to compute the resulting product by performing only three polynomial multiplications instead of four as shown in (3). $a(x)$, $b(x)$ and $c(x)$ are the three products that we have to calculate. The straightforward implementation of this algorithm consists of storing each of these three polynomials in extra memory and lastly add them up in the space for the $2N-2$ coefficients of the result. Since this algorithm can be applied recursively this leads to an excessive extra memory required. Instead, we use Algorithm 4 where all operations are performed over the output space. Figure 6 shows how the output is arranged between each of the three calls to the multiplication to be able to reutilize the space while getting the correct result. The left most chunk in the figure represents the lower k coefficients of the result, the next left chunk represent the coefficients that will be multiplied by x^k in the final result, and the next two those which will be multiplied by x^{2k}. Also, we consider all polynomials of N coefficients as $p(x) = p_0 + p_1 \cdot x^k$. The result of the algorithm is accumulated with the initial value $h(x)$.

$$\begin{aligned} f(x) * g(x) &= (f_0 + f_1 x^k) \cdot (g_0 + g_1 x^k) = f_0 g_0 + (f_0 g_1 + f_1 g_0) \cdot x^k + f_1 g_1 \cdot x^{2k} \\ &= f_0 g_0 + ((f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1) \cdot x^k + f_1 g_1 \cdot x^{2k} \\ &= a + (c - a - b) \cdot x^k + b \cdot x^{2k} \end{aligned} \tag{3}$$

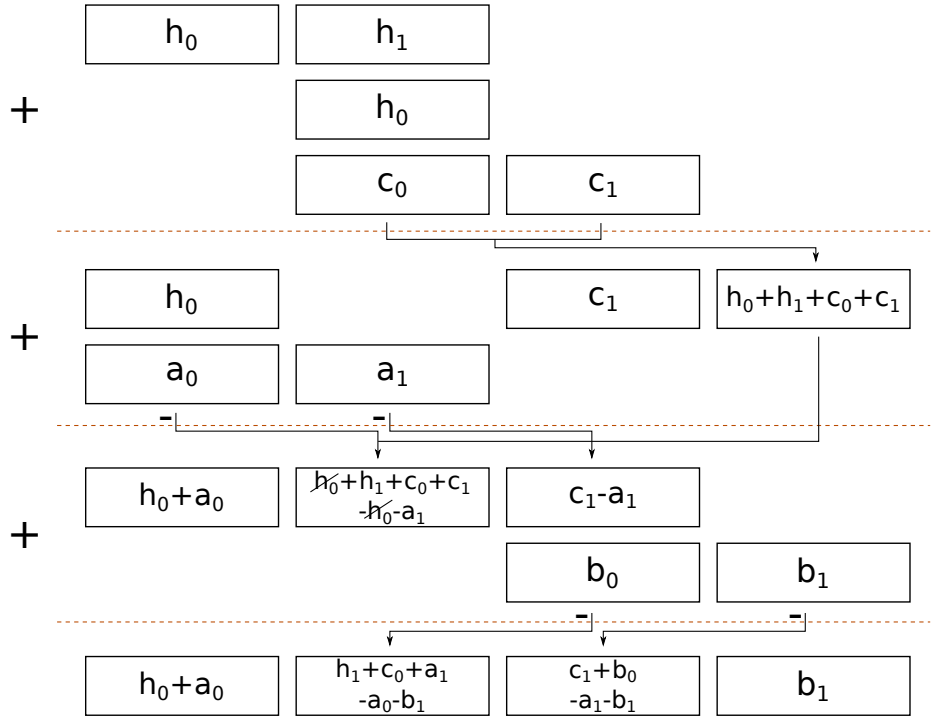

Figure 6: Details of the arrangements carried out over the output vector for Algorithm 4.

C Unrolled memory efficient Karatsuba

Algorithm 12: Memory efficient Karatsuba *kara_mem*

Input: Two polynomials $f(x)$ and $g(x)$ and their degree n
Output: $h(x) = h(x) + f(x) * g(x)$ of degree $2n - 1$

extra memory $tmp[n/4]$;
$k = n/2$
for $i = 0$ *to* $k/2$ **do**
  $h[k+i] = h[i] + h[k+i]$;
  $h[k+k/2+i] = h[k/2+i] + h[k+i] + h[k+k/2+i]$;
  $h[3k-1+i] = f[i] + f[k+i]$;
  $h[3k+k/2-1+i] = f[k/2+i] + f[k+k/2+i]$;
for $i = 0$ *to* $k/2$ **do**
  $tmp[i] = h[3k-1+i] + h[3k+k/2-1+i]$;
  $h[2k+k/2+i] = g[i] + g[k/2+i] + g[k+i] + g[k+k/2+i]$;
`poly_mul`$(tmp, h[2k+k/2], h[k+k/2], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[2k+k/2-1+i] = h[k+k/2+i] + h[2k+i]$;
  $tmp[i] = g[i] + g[k+i]$;
`poly_mul`$(tmp, h[3k-1], h[k], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[2k+i] = h[2k+i] - h[k+k/2+i]^{(*)}$;
  $h[k+k/2+i] = h[2k+k/2-1+i] - h[k+i]$;
  $tmp[i] = g[k/2+i] + g[k+k/2+i]$;
`poly_mul`$(tmp, h[3k+k/2-1], h[2k], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[k/2+i] = h[i] + h[k/2+i]$;
  $h[2k+i] = h[2k+i] - h[2k+k/2+i]^{(*)}$;
  $h[3k-1+i] = h[2k+i] + h[k+i]$;
  $h[3k+k/2-1+i] = h[k+k/2+i] - h[2k+i]$;
for $i = 0$ *to* $k/2$ **do**
  $h[k+k/2-1+i] = f[i] + f[k/2+i]$;
  $tmp[i] = g[i] + g[k/2+i]$;
`poly_mul`$(tmp, h[k+k/2-1], h[k/2], k/2)$
for $i = 0$ *to* $k/2-1$ **do**
  $h[k+k/2-1+i] = h[k/2+i] + h[k+i]$;
`poly_mul`$(f, g, h, k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[k+i] = h[k+i] - h[k/2+i]^{(*)}$;
  $h[k/2+i] = h[k+k/2-1+i] - h[i]$;
`poly_mul`$(f[k/2], g[k/2], h[k], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[k/2+i] = h[k/2+i] - h[k+i]$;
  $h[2k+i] = h[2k+i] - h[k+i] + h[k+k/2+i]^{(*)}$;
  $h[k+i] = h[3k-1+i] - h[i]$;
  $h[2k+k/2+i] = h[2k+i] + h[2k+k/2+i] - h[k+k/2+i]^{(*)}$;
  $h[k+k/2+i] = h[3k+k/2-1+i] - h[k/2+i]$;
  $h[3k+k/2-1+i] = f[k+i] + f[k+k/2+i]$;
  $tmp[i] = g[k+i] + g[k+k/2+i]$;
`poly_mul`$(tmp, h[3k+k/2-1], h[2k], k/2)$
for $i = 0$ *to* $k/2-1$ **do**
  $h[3k+k/2-1+i] = h[2k+k/2+i] + h[3k+i]$;
`poly_mul`$(f[k], g[k], h[2k], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[3k+i] = h[3k+i] - h[2k+k/2+i]^{(*)}$;
  $h[2k+k/2+i] = h[3k+k/2-1+i] - h[2k+i]$;
`poly_mul`$(f[k+k/2], g[k+k/2], h[2k], k/2)$
for $i = 0$ *to* $k/2$ **do**
  $h[2k+k/2+i] = h[2k+k/2+i] - h[3k+i]$;
  $h[3k+i] = h[3k+i] - h[3k+k/2+i]^{(*)}$;
  $h[k+i] = h[k+i] - h[2k+i]$;
  $h[2k+i] = h[2k+i] - h[3k+i]$;
  $h[k+k/2+i] = h[k+k/2+i] - h[2k+k/2+i]$;
  $h[2k+k/2+i] = h[2k+k/2+i] - h[3k+k/2+i]^{(*)}$;
return $h(x)$;
$^{(*)}$ skip last iteration