Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms

Jose Maria Bermudo Mera, Angshuman Karmakar, Suparna Kundu, Ingrid Verbauwhede

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

Abstract. In this paper, we introduce Scabbard, a suite of post-quantum key-encapsulation mechanisms. Our suite contains three different schemes Florete, Espada, and Sable based on the hardness of module- or ring-learning with rounding problem. In this work, we first show how the latest advancements on lattice-based cryptography can be utilized to create new better schemes and even improve the state-of-the-art on post-quantum cryptography.
We put particular focus on designing schemes that can optimally exploit the parallelism offered by certain hardware platforms and are also suitable for resource constrained devices. We show that this can be achieved without compromising the security of the schemes or penalizing their performance on other platforms.
To substantiate our claims, we provide optimized implementations of our three new schemes on a wide range of platforms including general-purpose Intel processors using both portable C and vectorized instructions, embedded platforms such as Cortex-M4 microcontrollers, and hardware platforms such as FPGAs. We show that on each platform, our schemes can outperform the state-of-the-art in speed, memory footprint, or area requirements.

Keywords: Post-quantum cryptography · Learning with rounding · Key-encapsulation mechanism · Lattice-based cryptography · Hardware implementations · FPGA · Cortex-M4 · AVX2

1 Introduction

Lattice-based hard problems started to gain traction in cryptography with the introduction of Regev's learning with errors (LWE) [Reg04] and Lyubashevsky *et al.*'s [LPR10] ring-learning with errors (RLWE) as an alternative to integer factorization and elliptic-curve based cryptosystems. However, the launch of the National Institute of Standards and Technology's (NIST) post-quantum standardization program [NIS17] undeniably imparted a fresh impetus to the development of lattice-based cryptography. The majority of the 80 initial submissions in this program were based on lattices. During the first phase of the NIST competition designers incorporated many fresh ideas into the design of lattice-based cryptography, e.g., the Falcon signature scheme [FHK+18] was designed based on Gentry, Peikert and Vaikuntanathan's framework [GPV07] for signatures instead of the more traditional Fiat-Shamir (with abort) [FS87, Lyu09] framework, Kyber [BDK+17], Saber [DKRV19], Dilithium [DKL+18] used module-lattices instead of more traditional standard or ideal lattices, Titanium [SSZ19] used the middle-product LWE [RSSS17] problem to construct a key-encapsulation mechanism (KEM) instead of LWE or RLWE, etc. During the latter phases, the cryptographic community is witnessing a substantial effort to improve the designs and implementations [KRS18, HOKG18, KBMSRV18, BUC19],

come out with new physical attacks [ADP18, BP18, PPM17], and find better concrete security estimates [APS15, DSDGR20]. Such efforts have enriched the knowledge of lattice-based cryptography to an unprecedented level.

The primary motivation of our work is to show that carefully crafted decisions motivated by innovations in the lattice-based cryptography during last couple of years can lead to very efficient designs of cryptosystems. We want to show that due to changes at design level it is possible to instantiate our schemes by using off the shelf hardware and software implementations with small adaptations only. We also show that it is possible to improve the design of existing schemes using these advancements. Finally, we want to design KEMs with a particular focus on practicality. Our schemes should be efficient on a wide range of hardware and software platforms. To bolster confidence in our schemes we refrain from assuming aggressive assumptions in our design decisions which have been shown to be vulnerable to various attacks during past couple of years. We only use design elements which have stood the test of time by going through rigorous security evaluations during the lifetime of the NIST's standardization effort and, thus, elicit high confidence. Furthermore, we take into account the state-of-the-art cryptanalysis and security estimation techniques while proposing concrete instantiations of our designs. We conclude this section by briefly summarizing our contributions as below.

1. We propose Scabbard, a suite of new lattice-based KEMs. Our first scheme, named Florete, is a ring-learning with rounding (RLWR) based KEM. We used one of the third round finalists of NIST's program Saber's hardware and software implementations with some modifications for an efficient implementation of Florete. Our results show that Florete is one of the fastest KEMs when compared to other finalist KEMs in the NIST's post-quantum standardization procedure.

2. The introduction of module-lattices [LS15] opened up a whole spectrum of new lattices to designers who were earlier left with only standard or ring-lattices. Although, there exist module-lattice based schemes such as Kyber [BDK+17] and Saber [DKSRV18], it is beneficial to explore other constructions. Here, we propose the first of its kind module-learning with rounding (MLWR) with small degree polynomials named Espada, the second KEM in our suite. Espada has been designed to exploit parallelism on hardware platforms and achieves the lowest memory footprint among all KEM finalist in the NIST's standardization process on software platforms.

3. The errors in learning with rounding (LWR) based schemes are generated by rounding elements of one number field to another. Since these errors influence the security of the KEM, it is important to estimate them properly. In this work, we properly formalize the distribution of such errors. We combine this with state-of-the-art cryptanalytic methods to propose improved parameters for Saber. We also suggest a new design choice for Saber. Being an MLWR based scheme, Saber is very flexible and scalable in terms of security and resource utilization. We show that incorporating our design choices further boosts these characteristics. We also show that using our parameters it is possible to improve the hardware designs in the state-of-the-art and reduce the key-sizes, and hence the required bandwidth of Saber. We name this modified Saber as Sable and include it as the third KEM in our suite.

4. We provide efficient software implementations optimized for general-purpose Intel processors and Cortex-M4 micro-controllers for all our schemes, and propose hardware architectures for accelerate them on field-programmable gate arrays. We compare our implementations with the state-of-the-art to demonstrate the efficiency of our schemes. All our implementations strictly avoid branching on secret data and run in constant-time. All our sources are publicly available[1].

[1] https://github.com/josebmera/scabbard

2 Preliminaries

We denote the set of integers $\{0, \ldots, q-1\}$ as $\mathbb{Z}_q$. We refer to the quotient ring $\mathbb{Z}_q[x]/(1+x^n)$ by $\mathscr{R}_q^n$ unless otherwise stated. In this work, the moduli p and q are power-of-two integers $(p < q)$. We denote the ring of $(l \times m)$-matrices over any ring R as $R^{l\times m}$ and the ring of l-length vectors over any ring R as R^l. Polynomials are denoted by lower case alphabets, vectors are denoted by bold lower case alphabets, and matrices are denoted by bold upper case alphabets. If $a \in \mathscr{R}_q^n$, then the scaling down operation $\lfloor\cdot\rceil_p : \mathscr{R}_q^n \to \mathscr{R}_p^n$ is defined by applying the rounding operator $\lfloor \frac{p}{q}(\cdot)\rceil$ to each coefficient of a and is extended to vectors by applying it to each element. We denote the uniform distribution as $\mathscr{U}$. The centered binomial distribution (CBD) is denoted by β_η, where the standard deviation is $\sqrt{\eta/2}$. Sampling according to β_η is realized by calculating $\sum_{i=0}^{\eta-1}(b_i - b'_i)$, where b_i and b'_i are pseudo-random bits. Random sampling from any set S according to a distribution χ is denoted by $\leftarrow \chi(S)$ and $\cdot$ represents the matrix-vector multiplication, vector-vector multiplication or polynomial multiplication depending on the context. The $\texttt{bits}(x, i, j)$ operator is a selection function that takes as input positive integers x, i, j with $i \geq j$ and outputs j consecutive `bits` of the positive integer x, ending at the i-th index, where the least significant bit (LSB) is the 1st index. It is extended to polynomials, vectors and matrices by applying it coefficient-wise.

2.1 Learning with errors and its variants

The learning with rounding (LWR) by Banerjee *et al.* [BPR12] is a variant of the well known learning with errors (LWE) problem introduced by Oded Regev [Reg04]. An LWE sample is of the form $(\mathbf{A}, \mathbf{b} = \mathbf{A}\cdot\mathbf{s} + \mathbf{e}) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^m$ whereas an LWR sample has the form $(\mathbf{A}, \mathbf{b} = \lfloor \frac{p}{q}\mathbf{A}\cdot\mathbf{s}\rceil = \lfloor \mathbf{A}\cdot\mathbf{s}\rceil_p) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_p^m$. Here, the error e is generated inherently because of the scaling from $\mathbb{Z}_q$ to $\mathbb{Z}_p$, where $p < q$. The decisional version of LWR problem states that it is hard to distinguish between the LWR samples and $(\mathbf{A}, \mathbf{u}) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_p^m$, where $\mathbf{s}$ is sampled from $\chi(\mathbb{Z}_q^n)$ for a specific distribution χ, $\mathbf{A}$ and $\mathbf{u}$ are sampled uniformly from $\mathbb{Z}_q^{m\times n}$ and $\mathbb{Z}_q^m$ respectively.

Similar to the Ring-LWE problem introduced by Lyubashevsky *et al.* [LPR10], the decisional version of Ring-LWR problem states that it is hard to distinguish between the samples of the form $(a, b = \lfloor \frac{p}{q}a\cdot s\rceil) \in \mathscr{R}_q^n \times \mathscr{R}_p^n$ from $(a, u) \in \mathscr{R}_q^n \times \mathscr{R}_q^n$, where s is sampled from $\chi(\mathscr{R}_q^n)$ for a specific distribution χ, a and u are sampled uniformly from $\mathscr{R}_q^n$.

Modules lattices [LS15] were introduced as a trade-off between standard and ideal lattices in terms of efficiency and security. The decisional version of Module-LWR problem states that it is hard to distinguish between the samples of the form $(\mathbf{A}, \mathbf{b} = \lfloor \frac{p}{q}\mathbf{A}\cdot\mathbf{s}\rceil) \in (\mathscr{R}_q^n)^{l\times l} \times (\mathscr{R}_p^n)^l$ from $(\mathbf{A}, \mathbf{u}) \in (\mathscr{R}_q^n)^{l\times l} \times (\mathscr{R}_p^n)^l$, where s is sampled from $\chi((\mathscr{R}_q^n)^l)$ according to the specific distribution χ, $\mathbf{A}$ and $\mathbf{u}$ are sampled uniformly from $(\mathscr{R}_q^n)^{l\times l}$ and $(\mathscr{R}_q^n)^l$ respectively.

The rank of the underlying matrices in these problems is n for LWR and RLWR and $l \times n$ for MLWR with very high probability. In the absence of efficient attacks that exploit the underlying algebraic structure to their advantage and when all other parameters such as q, p and χ are kept the same, the security of all cryptosystems based on these lattices is considered the same if the rank of their underlying matrices is the same. The structure of Module-LWR is more generic as we can convert it to Ring-LWR by making $l = 1$ and $n = n$, and to standard LWR by setting $l = n$ and $n = 1$. For the rest of this paper, we considered the structure of the Module-LWR problem as a generalized LWR problem.

2.2 LWR key-exchange (KEX) protocol

A generalized LWR based key-exchange (KEX) is shown in Protocol. 1. To accomplish this, we need another power-of-two modulus t such that $t < p < q$. Here, the function `gen` generates the public pseudo-random matrix $\mathbf{A}$ with the help of an extendable-output function `XOF` and a 256-bit random `seed`. Unlike the classic Diffie-Hellman [DH76] KEX, LWR or in general LWE based schemes may not end up with same keys. This is due to he fact that the difference between the final polynomials Alice (u) and and Bob (u') are not negligible. Hence we need an error correction scheme [Pei14, Din12] described in Sec. 2.4. A KEX is called IND-RND secure if the advantage of any adversary $\mathscr{A}$ to distinguish the key $k \in K$ (if K is the key space) generated by the KEX from a uniformly random chosen key $k' \in K$ is negligible. It can be proven that the generalized LWR based KEX as shown in Protocol 1 is IND-RND secure if $q/p \leq p/(2^B t)$. This proof closely follows the security proof of Saber [DKSRV18].

	Alice		Bob
1	$seed_{\mathbf{A}} \leftarrow \mathscr{U}(\{0,1\}^{256})$		
2	$r \leftarrow \mathscr{U}(\{0,1\}^{256})$		
3	$\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathtt{XOF}(seed_{\mathbf{A}})) \in (\mathscr{R}_q^n)^{l\times l}$		
4	$\mathbf{s} \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$		
5	$\mathbf{b} = \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_p \in (\mathscr{R}_p^n)^l$		
6		$\xrightarrow{(\mathbf{b}, \text{seed}_{\mathbf{A}})}$	
7			$r' \leftarrow \mathscr{U}(\{0,1\}^{256})$
8			$\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathtt{XOF}(seed_{\mathbf{A}})) \in (\mathscr{R}_q^n)^{l\times l}$
9			$\mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
10			$\mathbf{b}' = \lfloor \mathbf{A}^T \cdot \mathbf{s}' \rceil_p \in (\mathscr{R}_p^n)^l$
11			$u' = \mathbf{b}^T \cdot (\mathbf{s}' \mod p) \in \mathscr{R}_p^n$
12			$c' = \mathtt{HelpDecode}(u') \in \mathscr{R}_t^n$
13		$\xleftarrow{(\mathbf{b}', c')}$	
14	$u = \mathbf{b}' \cdot (\mathbf{s} \mod p) \in \mathscr{R}_p^n$		
15	$c = \mathtt{Decode}(u, c') \in \mathscr{R}_{2^B t}^n$		
16	$k = \mathtt{Encode}(c)$		$k' = \mathtt{Encode}(u')$
17	$\mathtt{key}_{Alice} = \mathtt{Hash}(k)$		$\mathtt{key}_{Bob} = \mathtt{Hash}(k')$

Protocol 1: A generalized key-exchange scheme based on LWR

Theorem 1. *LWR based KEX is IND-RND secure if* $q/p \leq p/(2^B t)$.

Proof. See Appendix A. □

2.3 CCA secure LWR based KEM

The LWR based KEX is a noisy Diffie-Hellman key-exchange [DH76] and can be transformed to an indistinguishable against chosen plaintext attack (IND-CPA) secure public-key encryption (PKE), analogous to the transformation from a Diffie-Hellman key-exchange to the IND-CPA secure ElGamal PKE scheme. In the PKE, the message is added or XORed with each coefficient of the key k' of Bob in the KEX. The correctness of the PKE scheme also depends on the equality of the keys k and k' used in the KEX scheme. LWR based KEX and LWR based PKE are equivalent in terms of security and correctness. It is very simple to show that LWR based PKE scheme is IND-CPA secure if the underlying KEX is IND-RND secure.

Jiang *et al*. [JZC+17] provided a version of the Fujisaki-Okamoto transformation [FO99] to convert an IND-CPA secure LWR based PKE to an indistinguishable against chosen chiphertext attack (IND-CCA) secure key-encapsulation mechanism (KEM), when the underlying PKE scheme is not perfectly correct. The authors also proved that if the underlying PKE scheme is $(1-\delta)$ correct, then the KEM based on it will be S post-quantum secure where $\delta \leq 2^{-S}$.

Following Jian *et al.*'s construction, we provide generic algorithms for IND-CCA secure LWR based KEM (`KeyGen`, `Encaps`, `Decaps`) in Alg. 1, 2, and 3. For example, if we set $n = 256$, $l = 3$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^3$, $\eta = 4$, $B = 1$ we will get the Saber KEM.

In these algorithms, H and $\mathcal{G}$ are hash functions. $\mathbf{h_1}$, h_2, and h_3 are constant polynomials with each coefficient set to $2^{(\epsilon_q-\epsilon_p-1)}$, $(2^{(\epsilon_q-\epsilon_p-1)} + 2^{(\epsilon_p-B-1)} - 2^{(\epsilon_p-\epsilon_t-1)})$, and $2^{(\epsilon_q-\epsilon_p-1)}$ respectively. Here, $\epsilon_q = \log_2 q$, i.e., $q = 2^{\epsilon_q}$, similarly $p = 2^{\epsilon_p}$ and $p = 2^{\epsilon_t}$. These are used to calculate the rounding operators $\lfloor\cdot\rceil_p$ and $\lfloor\cdot\rceil_t$. In `KeyGen` Alg. 1 $H(\mathtt{pk})$ is stored in the public key and the `Decaps` Alg. 3 returns a random value if it fails. These are the extra parts of this FO-transformation for achieving CCA security.

Algorithm 1: LWR.KEM.KeyGeneration

Data: *nil*
Result: $\mathtt{pk} = (seed_{\mathbf{A}}, \mathbf{b}), \mathtt{sk} = (\mathbf{s}, H(\mathtt{pk}), r)$

$seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$r \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathtt{XOF}(seed_{\mathbf{A}})) \in (\mathcal{R}_q^n)^{l\times l}$
$\mathbf{s} \leftarrow \beta_\eta((\mathcal{R}_q^n)^l)$
$\mathbf{b} = \mathtt{bits}(\mathbf{A}\cdot\mathbf{s} + \mathbf{h_1}, \epsilon_q, \epsilon_p) \in (\mathcal{R}_p^n)^l$ //Rounding
$\mathtt{pk} \leftarrow (seed_{\mathbf{A}}, \mathbf{b})$ $r \leftarrow_\$ \{0,1\}^{256}$
$\mathtt{sk} \leftarrow (\mathbf{s}, H(\mathrm{pk}), r)$
return $\mathtt{pk} = (seed_{\mathbf{A}}, \mathbf{b}), \mathtt{sk} = (\mathbf{s}, H(\mathrm{pk}), r)$

Algorithm 2: LWR.KEM.Encapsulation

Data: $\mathtt{pk} = (seed_{\mathbf{A}}, \mathbf{b})$
Result: $\mathtt{cipher\_txt} = (c', b'), \mathtt{key} = K$

$m' \leftarrow_\$ \{0,1\}^{256}$
$m = \mathtt{arrange\_msg}(m')$
$(K', r') \leftarrow \mathcal{G}(m||H(\mathtt{pk}))$
$r' \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathtt{XOF}(seed_{\mathbf{A}})) \in (\mathcal{R}_q^n)^{l\times l}$
$\mathbf{s}' \leftarrow \beta_\eta((\mathcal{R}_q^n)^l)$
$\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T\cdot\mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$ //Rounding
$u' = \mathbf{b}^T\cdot(\mathbf{s}' \mod p) \in \mathcal{R}_p^n$
$c' = \mathtt{bits}((u' + h_3 - 2^{\epsilon_p-B}m), \epsilon_p, (\epsilon_t + B)) \in \mathcal{R}_{2^B t}^n$ //HelpDecode
$K \leftarrow H(K', H(c'))$
return $\mathtt{cipher\_txt} = (c', b'), \mathtt{key} = K$

Algorithm 3: LWR.KEM.Decaps

Data: $\mathtt{pk} = (seed_{\mathbf{A}}, \mathbf{b}), \mathtt{sk} = (\mathbf{s}, H(\mathtt{pk}), r), \mathtt{cipher\_txt} = (c', \mathbf{b}')$
Result: $\mathtt{key} = K$

$u = \mathbf{b}'\cdot(\mathbf{s} \mod p) \in \mathcal{R}_p^n$
$m_1' = \mathtt{bits}((u + h_2 - 2^{\epsilon_p-\epsilon_t-B}m), \epsilon_p, B) \in \mathcal{R}_{2^B}^n$ //Decode
$m_1 = \mathtt{original\_msg}(m_1')$
$m_2 = \mathtt{arrange\_msg}(m_1)$
$(K_1', r_1') \leftarrow \mathcal{G}(m_2||H(\mathrm{pk}))$
$\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathtt{XOF}(seed_{\mathbf{A}})) \in (\mathcal{R}_q^n)^{l\times l}$
$\mathbf{s_1}' \leftarrow \beta_\eta((\mathcal{R}_q^n)^l)$
$\mathbf{b_1}' = \mathtt{bits}(\mathbf{A}^T\cdot\mathbf{s_1}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$ //Rounding
$u_1' = \mathbf{b}^T\cdot(\mathbf{s_1}' \mod p) \in \mathcal{R}_p^n$
$c_1' = \mathtt{bits}((u_1' + h_3 - 2^{\epsilon_p-B}m), \epsilon_p, (\epsilon_t + B)) \in \mathcal{R}_{2^B t}^n$ //HelpDecode
if $c' = c_1'$ **then**
  return $K = H(K_1', H(c'))$
else
  return $K = H(r, H(c'))$

2.4 Error correction mechanism

The error correction mechanism contains three functions `Encode`, `Decode`, `HelpDecode`. Here, the target is to establish the following relation. Let, u and u' be the polynomials computed by Alice and Bob respectively and u_i, u_i' be the i-th coefficient of the polynomial u and u'.

If $|u_i - u_i'| < \epsilon$, where ϵ is the error tolerance, then $\mathtt{Encode}(\mathtt{Decode}(u_i, \mathtt{HelpDecode}(u_i')))$ $= \mathtt{Encode}(u_i')$ with high probability. If we take $u_i' = x$ and $u_i = y$, then the functions `Encode`, `HelpDecode` and `Decode` are defined by $\mathtt{Encode}(x)$ = "first B bits of x", $\mathtt{HelpDecode}(x)$ = "next ϵ_t bits of x" and $\mathtt{Decode}(y, \mathtt{HelpDecode}(x)) = y - \mathtt{HelpDecode}(x)\frac{q}{2^{B+\epsilon_t}}$. These functions can be extended to polynomial by applying them coefficient-wise. It can

be shown that if the absolute value of error tolerance is bounded by $\frac{q}{2^{B+1}} - \frac{q}{2^{B+\epsilon_t+1}}$ then the above requirement is satisfied.

Theorem 2. *If $x = y + e$ and $|e| \leq \frac{q}{2^{B+1}} - \frac{q}{2^{B+\epsilon_t+1}}$, then*

$$\texttt{Encode}(x) = \texttt{Encode}(\texttt{Decode}(y, \texttt{HelpDecode}(x))).$$

Proof. See Appendix B. □

2.5 Polynomial multiplication

There are two efficient algorithms for multiplying two polynomials $a, b \in \mathscr{R}_q^n$, the number theoretic transform (NTT) [Pol71] which runs in $O(n \log n)$ and the Toom-Cook [Too63, Coo66, KO62] based polynomial multiplication that runs in $O(n^{1+\epsilon}), 0 < \epsilon < 1$. While the NTT is faster, it forces few constraints on the degree of the polynomial n and modulus q. Many RLWE and Module-LWE schemes [BDK+17, ADPS16, DKL+18] use this polynomial multiplication.

We will only discuss Toom-Cook multiplication or specifically Toom-Cook k-way here since it is the most relevant to our work. Given $a, b \in \mathscr{R}_q^n$ a pre-processing stage, evaluation, is applied to create a vector of length $2k-1$ from each of a and b, where each element in the vector is a polynomial of length n/k. Each element from each vector can be further split into smaller polynomials by applying Toom-Cook k-way evaluation recursively until the polynomials are small enough to be multiplied with the corresponding polynomial in the other vector by the quadratic complexity schoolbook multiplication algorithm. After the multiplication stage, the Toom-Cook k-way interpolation is applied recursively on the results to get the resulting polynomial $c = a \cdot b$. The Toom-Cook 3-way and Toom-Cook 4-way algorithms are described in Alg. 4 and 5 in Appendix G. For more details on Toom-Cook multiplication we refer the interested reader to [Ber01, MKV20].

3 Our suite of LWR based KEMs

This section describes our Scabbard KEM suite. All of our schemes follow the generic KEM=(`KeyGen`, `Encaps`, `Decaps`) constructions as shown in Alg. 1, 2, and 3 respectively. Only the ring/module parameters (n, l), moduli $(\epsilon_q, \epsilon_p, \epsilon_t)$, encoding parameter (B), CBD parameter (η) and polynomial multiplication change in each scheme. Hence, in the description of our schemes we will only discuss these parameters that are unique to each scheme and their implications. We discuss in detail our design rationale, implementation strategies, challenges and our approaches to overcome them. We first discuss the shape of rounding-errors of LWR based cryptosystems which is very crucial to our designs.

3.1 Rounding error : discrete vs. continuous uniform distribution

As we have discussed before, the errors in LWR based cryptosystems are generated inherently. A series of recent LWR based cryptosystems such as (Round2 [BBG+17], Saber [DKRV19], Lizard [CKLS18]) considered this error as continuous uniform in the interval $(-q/2p, q/2p]$. The following Theorem 3 shows that this error distribution is discretely uniform rather than continuous uniform as assumed earlier.

Consider LWR samples of the form $(\mathbf{A}, \mathbf{b} = \lfloor \mathbf{A}.\mathbf{s} \rceil_p) \in (\mathscr{R}_q^n)^{l \times l} \times (\mathscr{R}_p^n)^l$, with $n \geq 1$ and $q > p \geq 2$. We can write $\mathbf{b} = (b_1, b_2, \ldots, b_l)$, where $b_i = (b_i^j)$.

Theorem 3. *Each coefficient of every polynomial of the rounding error vector follows a discrete uniform distribution over the set $\{-q/2p, \ldots, q/2p - 1\}$.*

Proof. Let us consider the j-th coefficient of i-th polynomial of rounding error is e and $\texttt{bits}(b_i^j, \epsilon_{q-p}, \epsilon_{q-p}) = \lambda$. Then $e = b_i^j - \lfloor (q/p)\lfloor b_i^j \rceil_p \rceil = \lfloor (q/p) f \rceil$,

$$\text{where } f = \begin{cases} -1+\frac{1}{2} & \text{if } \lambda = \frac{q}{2p} \\ -1+\frac{1}{2}+\frac{1}{2^{(\epsilon_{q-p})}} & \text{if } \lambda = \frac{q}{2p}+1 \\ \vdots & \\ -1+\frac{1}{2}+\cdots+\frac{1}{2^{(\epsilon_{q-p})}} & \text{if } \lambda = \frac{q}{p}-1 \\ 0 & \text{if } \lambda = 0 \\ \frac{1}{2^{(\epsilon_{q-p})}} & \text{if } \lambda = 1 \\ \frac{1}{2^{((\epsilon_{q-p})-1)}} & \text{if } \lambda = 2 \\ \frac{1}{2^{((\epsilon_{q-p})-1)}}+\frac{1}{2^{(\epsilon_{q-p})}} & \text{if } \lambda = 3 \\ \vdots & \\ \frac{1}{2^2}+\cdots+\frac{1}{2^{(\epsilon_{q-p})}} & \text{if } \lambda = \frac{q}{2p}-1 \end{cases} \text{.So, } e = \begin{cases} -\frac{q}{2p} & \text{if } \lambda = \frac{q}{2p} \\ -\frac{q}{2p}+1 & \text{if } \lambda = \frac{q}{2p}+1 \\ \vdots & \\ -1 & \text{if } \lambda = \frac{q}{p}-1 \\ 0 & \text{if } \lambda = 0 \\ 1 & \text{if } \lambda = 1 \\ 2 & \text{if } \lambda = 2 \\ 3 & \text{if } \lambda = 3 \\ \vdots & \\ \frac{q}{2p}-1 & \text{if } \lambda = \frac{q}{2p}-1 \end{cases}.$$

As $\lambda = \texttt{bits}(b_i^j, \epsilon_{q-p}, \epsilon_{q-p})$, then $\Pr[\lambda = \lambda'] = 1/2^{(\epsilon_{q-p})} = p/q$, $\forall \lambda' = \{0, 1, \ldots, (\frac{q}{p}-1)\}$. Therefore, $\Pr[e = e'] = p/q$, $\forall e' = \{-q/2p, -q/2p+1, \ldots, q/2p-1\}$. Hence, e follows a discrete uniform distribution over the set $\{-q/2p, -q/2p+1, \ldots, q/2p-1\}$ □

While evaluating the security and failure probability of any LWE based cryptosystem, the variance of the error plays a very crucial role. The variance of the continuous uniform distribution and discrete uniform distribution are $\frac{q^2}{12p^2}$ and $\frac{q^2-p^2}{12p^2}$ respectively. As we can see $\frac{q^2-p^2}{12p^2} < \frac{q^2}{12p^2}$, considering the rounding error as continuous uniform overestimates the error distribution and consequently the concrete security estimation. Security of lattice-based cryptosystems is proportional to the ratio of standard deviation of error and moduli. Hence, to maintain security we have to decrease the moduli to compensate for the lower standard deviation of error. The parameter calculation of lattice-based cryptography is an optimization problem where the modulus, rank of the lattice, and standard deviation are the control variable. Whereas security and failure probability are the objective function. The standard procedure [ADPS16, BDK+17] to solve this problem is to exhaustively search over a wide range of control variables and choosing options which best satisfiy the requirements. We have followed the same procedure to find parameters of our schemes. We have considered the above observation during the concrete security estimation of our cryptographic schemes in Sec. 4.1.

3.2 Florete: Ring-LWR based KEM

Our primary focus while designing Florete was to maximally reuse the already very efficient hardware architectures and software modules [KRS18, KBMSRV18, RB20, MKV20] that have been developed for Saber for a more efficient KEM without compromising the security.

Since the introduction of binomial distributions in lattice-based cryptography, the polynomial multiplication has become the most computationally expensive operation in lattice-based cryptography. Although, in some platforms such as Cortex-M4 the pseudo-random number generation can take upto 50% of the total execution time [KRS18]. Due to our choice of moduli we are unable to use asymptotically faster number theoretic transform (NTT) based polynomial multiplications without using a larger NTT friendly prime (discussed in Appendix C). Hence, we resort to generic Toom-Cook polynomial multiplications. Below we describe the fundamental building blocks of Florete.

Polynomial multiplication: For our efficient implementation of Florete, we fix our quotient ring $\mathcal{R}_q^n$ as $\mathbb{Z}_q[x]/(x^{768} - x^{384} + 1)$. Now, while multiplying two polynomials

$a, b \in \mathscr{R}_q^n$ during `KeyGen`, `Encaps`, and `Decaps` we first apply a Toom-Cook 3-way evaluation on a and b. This splits both of them into $2 * 3 - 1 = 5$ polynomials of length 256 each. To multiply these 256 length polynomials we use the efficient hardware and software routines to perform 256×256 polynomial multiplications for Saber. Further we can join these results using Toom-Cook 3-way interpolation to get the result $c = a \times b \in \mathscr{R}_q^n$ after reduction by $(x^{768} - x^{384} + 1)$. Since the computational cost for Toom-cook 3-way evaluation and interpolation are small (as shown in Alg. 4 in Appendix. G) the time to perform 5 individual 256×256 polynomial multiplications is very close to the time to perform one 768×768 polynomial multiplication using our strategy. Further, as we are working in RLWR, our underlying ideal lattice can be represented by a single public polynomial of length 768 whereas for Saber the underlying module-lattice needs 9 polynomials of length 256 each. A comparison of the required randomness and the number of 256×256 polynomial multiplications to generate the LWR samples in Florete and in Saber is shown in Table. 1. We can see from this table that Florete gains in efficiency compared to Saber in both number of multiplications and pseudo-random number generation.

Name	Pseudo-random number (Bytes)	256×256 multiplications		
		KeyGen	Encaps	Decaps
Florete	1152	5	10	15
Saber	4512	9	12	15

Table 1: Required pseudo-random bytes for generating public matrix ($\mathbf{A}$) and secret vector ($\mathbf{s}$), and number of 256×256 multiplications in Saber and Florete.

However, there is a small caveat in this arrangement. We want the coefficients of our polynomials to fit within 16 bits for efficient multiplication in vector processors, small microcontrollers, or FPGAs. While applying Toom-Cook interpolations we often need to divide the field elements by r, where $r = 2^d \cdot m$ with $gcd(m, 2) = 1$. Since we are working in the power-of-two finite fields there exists no inverse of $r = 2^d \cdot m$ where $d \geq 1$. To overcome this while dividing by r, first multiply the number by inverse of m in the field followed by right shift of d bits. Thus, if we limit ourselves to 16-bit word length then it means our ϵ_q can not be more than $16 - d$ bits long for correct multiplication in $\mathscr{R}_n^q$. The maximum value of d in Toom-Cook 3-way and Toom-Cook 4-way interpolation is $d = 1$ and $d = 3$ respectively. Hence, to combine these two algorithms according to our strategy our ϵ_q cannot be more than 12 bits long. Note that, in Saber's design $\epsilon_q = 13$, therefore this combination of multiplications do not work if we use Saber's parameters. Here, we utilize our observation on Sec. 3.1 to reduce the standard deviation of the error and reduction of ϵ_q as a compensation to achieve our goal of $\epsilon_q \leq 12$. As we can see from Table 2, this reduces the post-quantum security of Florete than Saber by 12 bits but it is still high enough to qualify for NIST security level 3.

Error correction and encoding: Schemes like Round5 [BGML+18, BBG+17], LAC [LLZ+18] used error-correcting codes to reduce their failure probabilities of their KEMs. But these schemes open up many avenues of side-channel attacks [GMR20, DTVV19, GJY19, SC19, Son19]. This was one of the biggest reason for them not to qualify to the NIST's post-quantum standardization final round [AASA+17] . We used the error correction mechanism shown in Sec. 2.4 which is also used by Saber. In this mechanism, the size of the second rounding modulus t is proportional to the maximum error that can be corrected. As we need a very low failure probability, ϵ_t should be large. And as $\epsilon_t < \epsilon_p < \epsilon_q$ this imposes a limit on ϵ_p and ϵ_q as well. Here, we are working in ring $\mathscr{R}_q^n$ with $n = 768$, with the only 256 bits of secret payload (m'). We can set $B = 1$ and use each coefficient of our polynomial to embed each bit of secret with repetitions as $\texttt{arrange\_msg}(m') = m'||m'||m'$. To recover the message we can take a majority vote as

shown below

$$\texttt{original\_msg}(m_1[b]) = \begin{cases} 0 & \text{if } m_1'[b] + m_1'[b+256] + m_1'[b+512] \leq 1 \\ 1 & \text{else} \end{cases}.$$

As the error tolerance is increased due to the use of repetition we can reduce ϵ_t without increasing the failure probability. This consequently helps to reduce ϵ_p and ϵ_q further.

Security levels: We have so far described Florete to target a NIST security level 3. Our strategy can be extended to provide a security level 1 version with $n = 512$ and using Karatsuba [KO62] to split the polynomial into three polynomials of length 256 for the multiplications. A security level 5 version can be provided with $n = 1024$ and using Toom-Cook 4-way to split the actual polynomial into 7 polynomials of length 256. We leave instantiation of different security levels of Florete as a future work. We provide the full parameter list of Florete in Table 2. Following the works of Alkim *et al.* [ABC19] and Chung *et al.* [CHK+20] it is possible to improve the speed of Florete further by using a larger NTT friendly prime. We discuss this in Appendix C.

3.3 Espada: Module-LWR based KEM

The fundamental motivation behind designing our Module-LWR based KEM Espada was to have a scheme that is extremely parallelizable and has a small memory footprint in resource-constrained devices. Overall we also keep the performance on other platforms within the practical limits. As before we aim for ≥ 128 bits of post-quantum security.

If we look carefully, cryptosystems based on module-lattices are very suitable for parallel implementation. To recapitulate, in module-lattice based cryptosystems we need multiplications of the form $\mathbf{A} \cdot \mathbf{s}$ or $\mathbf{b} \cdot \mathbf{s}$ where $\mathbf{A} \in (\mathscr{R}_q^n)^{l \times l}$ and $b, s \in (\mathscr{R}_q^n)^l$. On a detailed note, these multiplications are basically multiplications of $a, b \in \mathscr{R}_q^n$ which can be performed in parallel. Unfortunately, due to the large size of n it is very costly in terms of area requirement to have multiple instances of polynomial multipliers. As an example, for Saber, where $n = 256$, the recent compact implementation by Bermudo Mera *et al.* [MTK+20] first splits each 256×256 polynomial multiplication to 7 64×64 polynomial multiplications which are then performed in parallel. This implementation avoids the 2-levels of Karatsuba multiplication as 64×64 schoolbook multiplication is already very fast on the target hardware platform. The whole 256×256 multiplier requires 28 DSP units, which is a more scarce resource than LUTs or FFs on FPGAs, and creating multiple instances of the 256×256 would rapidly exhaust it. Another implementation by Roy *et al.* [RB20] focuses on high-speed implementation but it again requires prohibitively high area for parallel instantiations. Lastly, the implementation by Dang *et al.* [DFAG19] uses such a high number (256) of DSP units that even a single instance of the multiplier is only suitable for the most powerful FPGAs like the UltraScale+ family by Xilinx.

Therefore, if we make n smaller we can easily exploit the parallelism reducing the cost of creating multiple instances of multipliers. Depending on the value n and the implementation philosophy, one can either use a compact multiplier inspired by the small 64×64 multipliers in [MTK+20] or an approach based on the fast schoolbook multiplier in [RB20]. As n is small both of them will be very fast and will require very low area. In this way the multiple instances of $n \times n$ polynomial multipliers can perform the multiplications in batch. This is explained in Fig. 1, where we compare this approach to the use of small polynomial multipliers in parallel after applying Toom-Cook to break down a larger multiplication. Furthermore, in implementations of module-lattice based cryptosystems, the memory footprint is proportional to the size of one polynomial thanks to the *just-in-time* matrix generation and other techniques developed in the context of the NIST PQC competition [KBMSRV18, KRS18]. Considering this, we have optimally chosen n in Espada as 64. Keeping n small has another benefit. As the rank of the module-lattices

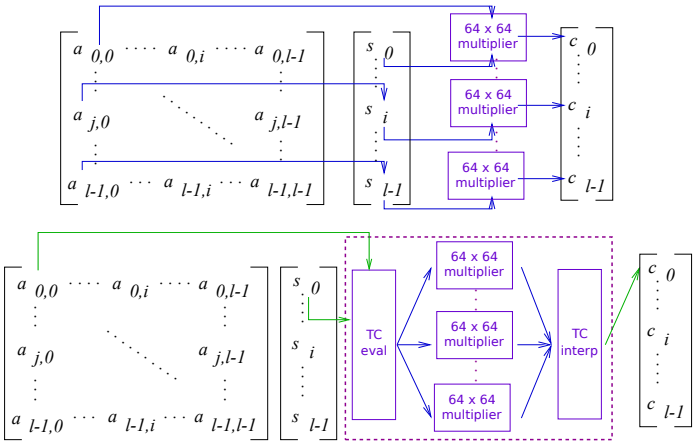

Figure 1: Comparison of parallel polynomial multiplication in Espada (top) with polynomial multiplication in Saber (bottom). The lines in blue and green denotes parallel and serial execution respectively. The components inside the boxes are implemented on hardware.

are multiples of n and the security is dependent on the rank of underlying matrix, having larger n often overshoots the security target. However, for small n we can have fine-grain control over security.

Encoding and error-correction: As our n is small and we are still considering a secret message payload of 256 bits, we set $B = 4$, i.e., we embed 4 secret bits in a single coefficient of the polynomial. However, according to Theorem 2, having a large B reduces the ability of the amount of error that can be corrected in our scheme. To compensate this, we have to increase ϵ_t. As $\epsilon_t < \epsilon_p < \epsilon_q$, this further requires a larger ϵ_p and ϵ_q to achieve the desired failure probability of $\leq 2^{-128}$.

Polynomial multiplication: As can be seen in Table 2, our modulus is 15 bits long. Hence, as discussed in Sec. 3.2, in order to limit ourselves within 16 bits of word-length, we cannot use algorithms such as Toom-Cook 3-way or 4-way multiplications. Hence, for software implementations we use two levels of Karatsuba to split each 64×64 polynomial multiplication into 9 16×16 polynomial multiplications. Also, in software implementation Toom-cook 4-way algorithm takes almost same time as 2-level karatsuba due to the interpolation and evaluation overhead. For hardware implementations we use a different approach which is described in Sec. 5. Further, as our $l = 12$ is quite large compared to other module-lattice based schemes, we use the lazy interpolation polynomial multiplication proposed in [MKV20]. As l is large we can reduce a lot of overhead for polynomial multiplication using this technique. We also use the optimized assembly routine from Kannwischer *et al.* [KRS18] for our microcontroller implementation.

Others: As we have reduced n, which in turn requires larger l and q for sufficient security and failure probability, we need to generate more pseudo-random numbers. Also we need to perform more 64×64 polynomial multiplications compared to Saber. Despite this, as shown in Sec. 4.2 our software implementation is quite fast for both portable C and micro-controller implementation. Similar to Florete, we can instantiate two other variants of Espada satisfying NIST security level 1 and 5 by increasing or decreasing l. Also, it might be possible to create a MLWE based KEM that can be instantiated with $n = 64$ and NTT friendly parameters using similar strategies to the ones explained here.

3.4 Sable: Alternate Saber

Sable is the third lattice-based KEM in our suite. As discussed in section Sec. 3.1, the Saber design used rounding error as continuous uniform distribution rather than discrete

uniform distribution. Due to a different standard deviation of error, we have to readjust other parameters, i.e., ϵ_p, ϵ_q and η to ensure that there is no significant drop in security. The updated parameters can be found in Table 2. We describe the rationale behind our choices below.

Secret distribution: We sample our secret values from the centered binomial distribution with $\eta = 1$ that means secret coefficients can be $-1, 0, 1$ only. This enables very fast multiplication in the platforms where multiplications are costlier than addition and subtraction such as MSP430 microcontrollers as the multiplication instructions can be replaced by additions and subtractions only. A recent hardware implementation has been proposed utilizing the small values of the secret [RB20]. We show in Sec. 5.3 that our parameters can further improve the performance and area of that hardware implementation. Furthermore, due to our choice $\eta = 1$, the secret can be stored using only 2-bits per coefficient. This results in a smaller memory requirement for Sable. Please note that here we have refrained from aggressive choices of secret distributions such as fixing the hamming weight of secret polynomials like Round5 [BGML+18], using any error correcting code to reduce failure probability like LAC [LLZ+18] or fixing the weight of the secret vector [BCLvV16]. We have stuck to binomial distribution to prevent the adversary from gaining any additional advantage due to the secret distribution. We discuss security implications in more details in Sec. 4.1. Currently, Saber team has produced another version called uSaber with 2 bits of uniform secret due to its advantage in implementation. This is very similar to our choice if we consider signed-bit representation.

New design choice: In Saber, we perform polynomial multiplications with the form of $a \cdot s$, where a is random in $\mathcal{R}_q^n$ or $\mathcal{R}_p^n$ and s sampled according to the distribution β_η. Saber has $\eta = 5, 4, 3$ for LightSaber, Saber and FireSaber respectively. We realized that it is more beneficial to keep η equal across all variants and varying ϵ_q and ϵ_p instead of keeping ϵ_q and ϵ_p same and varying η as done in Saber. Since secrets have a particular distribution, it is easy to exploit this distribution for every efficient implementation. Hence, if η is kept same for different variants, the multiplier can be heavily optimized and used in all variants. Since the distribution of a is random in $\mathcal{R}_q^n$ or $\mathcal{R}_p^n$, it is difficult to exploit the distribution of a for fast multiplication. In this case, the multiplier can be optimized for the maximum value of ϵ_q and ϵ_p only. This will work fine since ϵ_p and ϵ_q are power-of-two numbers. The recent implementation [RB20] for fast implementation of Saber exploits the special structure of the secret. However, they had to create additional hardware to support different η values. Since Saber designers stresses on the flexibility of the design, we think our design provides more flexibility than the original design. In addition to the fast polynomial multiplications, keeping a small η equal for all variants is also beneficial in masking. A recent paper on masking Saber [BDK+20] mentioned that the binomial sampler requires the most complex algorithms for masking among all the components. It also becomes more expensive for larger η. Indeed easier masking has been cited as the main reason for proposal of uSaber in the Round-3 submission of Saber [BMD+20] and a smaller η offers advantages for masking. We note that Kyber adopted the same design choice in their Round-2 submission [ABD+19].

4 Concrete instantiations

4.1 Security estimation

Like for other public-key cryptosystems, the concrete security of lattice-based cryptosystems is evaluated by calculating the time required by the best-known algorithm to solve the underlying computationally hard problem. For lattice-based cryptosystems this accounts to estimating the time to solve the underlying shortest vector problem using the block Korkine-Zolotarev [CN11, SE94] algorithm.

The state state-of-the-art solution which is used by almost all lattice-based schemes is using the *LWE-estimator* framework provided by Albrecht *et al.* [APS15]. Given $(n, q, \sigma_e, \text{secret distribution})$ where σ_e is the standard deviation of the error of a lattice based cryptosystem, this framework can compute the concrete security by estimating the run-time of all possible methods to solve the underlying hard lattice problem. Kindly note that, this estimator always considers error distribution as Gaussian distribution. Since the proposal of Applebaum *et al.* [ACPS09] to sample the secret from the same distribution as the errors most LWE based cryptosystems use $\sigma_s = \sigma_e$. However, in most LWR based cryptosystems [DKRV19, BBG+17, CKLS18] due to the rounding errors $\sigma_s < \sigma_e$. In this case, determining the concrete security of an LWR based cryptosystem using this framework while considering $\sigma_e = \sigma_s$ will lead to overestimation of the security of the scheme [2].

A new toolkit *leaky-LWE-Estimator* by Dachman-Soled *et al.* [DSDGR20] has been published recently to attack and estimate the hardness of the underlying LWE problem with side information. *Leaky-LWE-Estimator* considers (n, q, D_e, D_s) as input, where D_e, D_s are error distribution and secret distribution of a lattice-based cryptosystem respectively and outputs the security of that cryptosystem. Moreover, in this estimator, we have the flexibility to consider the error distribution of a cryptosystem as a discrete uniform distribution.

Since, any adversary instead of trying to solve the original LWR instance $\mathbf{A}, \mathbf{b} = \mathbf{A}\cdot\mathbf{s}+\mathbf{e}_r$ can solve the easier instance $(\mathbf{A}^{-1}, \mathbf{A}^{-1}\cdot\mathbf{b} = \mathbf{s}+\mathbf{A}^{-1}\cdot\mathbf{e}_r)$, here e_r is the inherently generated rounding error. Therefore, to avoid the problem of overestimating the concrete security, we have estimated the security of the scheme using Ducas *et al.*'s framework by considering minimum of (n, q, D_e, D_s) and (n, q, D_s, D_e) estimations.

As can be seen in Table 2, we have considered CBD with $\eta = 1$. This means our secrets can have values in the set $\{-1, 0, 1\}$. Recently, few attacks [GMR20, DTVV19, GJY19, DDGR20] have been proposed on schemes that have considered some aggressive secret distributions mainly to reduce failure probabilities, such as considering fixed hamming weight binomial distribution as secret distribution with an error correcting code (like LAC [LLZ+18]) or fixed the number of 1's and -1's of the ternary secret polynomial (like Round5 [BGML+18]). Although, Dachman-Soled *et al.* [DDGR20] proved that if the secret has a fixed number of ± 1 without knowing the exact amount of 1 and -1 (as in NTRU Prime [BCLvV16]), then the loss of security is negligible. Chen *et al.* [CCLS20] have also studied the some special ternary distributions and their security implications on lattice-based cryptography. To avoid adverse security implications we have refrained from taking any of such aggressive assumptions and used the standard binomial distribution where there is no fixed limit on the numbers of -1, 1, or 0. To the best of our knowledge, there does not exist any attack which can take advantage from binomially distributed secret distribution with $\eta = 1$.

4.2 Parameters and performance

We compare parameters of Scabbard with Saber in Table 2. As we can see, for similar security levels, all the variants of Sable improve the key sizes of Saber. Further, if we consider the bandwidth usage of each scheme, i.e., the combined size of public-key and ciphertext we can see that the bandwidth usage of Florete (2048 bytes) is slightly smaller than Saber (2080 bytes) despite being an ideal-lattice based scheme. The bandwidth of Espada (2584 bytes) is expected to be higher than Saber due to larger moduli. However, this increase is less than 25%.

Software performance: We have compiled our portable C implementations and

[2] Due to this reason the concrete security of Saber was overestimated in their previous submissions [DKRV19, DKSRV18]. This has been rectified in the round-3 submission [BMD+20]

Scheme Name	Ring/Module Parameters	PQ Security	Failure probability	Moduli	CBD (β_η)	Encoding	Key sizes for KEM (Bytes)
Florete	n: 768 l: 1	2^{157}	2^{-131}	ϵ_q: 10 ϵ_p: 9 ϵ_t: 3	$\eta = 1$	B=1	Public key: 896 Secret key: 1152 Ciphertext: 1248
Espada	n: 64 l: 12	2^{128}	2^{-167}	ϵ_q: 15 ϵ_p: 13 ϵ_t: 3	$\eta = 3$	B=4	Public key: 1280 Secret key: 1728 Ciphertext: 1304
LightSable	n: 256 l: 2	2^{104}	2^{-139}	ϵ_q: 11 ϵ_p: 9 ϵ_t: 2	$\eta = 1$	B=1	Public key: 608 Secret key: 800 Ciphertext: 672
Sable	n: 256 l: 3	2^{169}	2^{-143}	ϵ_q: 11 ϵ_p: 9 ϵ_t: 4	$\eta = 1$	B=1	Public key: 896 Secret key: 1152 Ciphertext: 1024
FireSable	n: 256 l: 4	2^{203}	2^{-208}	ϵ_q: 11 ϵ_p: 10 ϵ_t: 2	$\eta = 1$	B=1	Public key: 1312 Secret key: 1632 Ciphertext: 1376
LightSaber	n: 256 l: 2	2^{107}	2^{-120}	ϵ_q: 13 ϵ_p: 10 ϵ_t: 2	$\eta = 5$	B=1	Public key: 672 Secret key: 992 Ciphertext: 736
Saber	n: 256 l: 3	2^{172}	2^{-136}	ϵ_q: 13 ϵ_p: 10 ϵ_t: 3	$\eta = 4$	B=1	Public key: 992 Secret key: 1440 Ciphertext: 1088
FireSaber	n: 256 l: 4	2^{236}	2^{-165}	ϵ_q: 13 ϵ_p: 10 ϵ_t: 5	$\eta = 3$	B=1	Public key: 1312 Secret key: 1760 Ciphertext: 1472

Table 2: Comparison of Scabbard suite with Saber

vectorized implementations using advanced vector instructions (AVX2) using GCC 6.5 having optimization flags `-O3` and `-fomit-frame-pointer` enabled on an Intel(R) Core(TM) i7-6600 CPU running at 2.60GHz. We also disabled hyperthreading, turbo-boost and multicore support as per standard procedure. For our Cortex-M4 implementations, all performance and memory measurements were taken using the easy to use the framework provided in [KRSS] on an STM32F4DISCOVERY board running at 24 MHz.

As we can observe in Table 3 and 4, Florete has better performance in all software implementations. In the Cortex-M4 platform, it performs better than Saber by 48% and 25%, 14% in `KeyGen`, `Encaps` and `Decaps` respectively. The `KeyGen`, `Encaps` algorithms of Florete perform 42%, 11% faster than Kyber respectively. Only the `Decaps` algorithm is 11% slower in Florete than Kyber. It requires larger memory due to the ring structure. However, it still requires less memory compared to NTRU [CDH+19] except `Encaps`.

Espada has the lowest memory footprint among all KEMs. It requires, 61%, 67%, 69% and 11%, 28%, 33% less memory than Saber and Kyber for `KeyGen`, `Encaps` and `Decaps` respectively. Espada (17856 Bytes) requires almost 4 times more pseudo-random numbers than Saber (4512 Bytes) and almost twice 64×64 polynomial multiplications (for Espada 468 and for Saber 252). In software implementations, Keccak algorithm takes more than 50% of the execution time to generate the matrix and secret vector. Despite of all these disadvantages, the running time of this scheme in software is approximately 2.5 times slower than the Saber in the worst case, which we still believe that is suitable for practical scenarios. On the other hand, if one uses faster pseudo-random number generators than Keccak in software such as AES-CTR mode then better performances can be achieved. Kyber uses NTT multiplication which is an in-place algorithm in terms of memory. Despite this, Espada needs less memory than Kyber. Lastly, the performance and memory requirements of Sable are better than Saber in every platform. For a better insight into these results we have provided a breakdown of clockcyles on Cortex-M4 platform in Appendix D.

We have also included a concurrent work by Chung *at al*. [CHK+20] which employs

NTT to perform polynomial multiplication in Saber in the comparison. We have described how this strategy can be applied to Sable in Appendix C.2. Table 8 shows that even our preliminary implementation of NTT-Sable has better efficiency than NTT-Saber using this method. Although using a slower polynomial multiplication routine than NTT-Saber, we can see Florete still has better efficiency than NTT-Saber except `Decaps` (slower by 14%). We have sketched how Florete can also be implemented using NTT in Appendix C. We firmly believe when employed this method Florete will outperform all the KEMs in all of `KeyGen`, `Encaps`, and `Decaps`.

Scheme Name	Security level	C (X1000 clock cycles)			AVX (X1000 clock cycles)		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	Medium	86	147	193	58	87	97
Espada	Medium	334	354	350	258	273	267
Sable	Low	79	106	116	54	64	58
	Medium	152	186	207	80	95	89
	High	240	285	308	125	143	144
Saber	Low	88	129	120	66	76	73
	Medium	159	201	215	107	118	112
	High	265	315	338	149	173	165
Kyber	Low	154	209	250	271	335	310
	Medium	252	324	365	537	594	557
	High	400	515	566	874	969	907
NewHope	Low	136	205	231	76	121	124
	High	265	404	448	139	224	233

Table 3: Performance comparison in portable C and AVX2 implementation of Scabbard with other lattice-based KEMs.

Scheme Name	Cortex-M4 performance (X1000 clock cycles)			Cortex-M4 memory (bytes)		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	439	814	953	18252	18420	18420
Espada	2343	2568	2497	2896	2120	2000
Sable	745	1004	1028	6184	5992	5496
Saber†	846	1098	1112	7488	6560	6568
NTT-Saber [CHK+20]	658	864	835	27440	29080	30176
Kyber*	763	923	862	3276	2964	2988
NTRUhrss701*	153104	377	869	27560	7400	20552
NTRUhps2048677*	143734	821	815	28504	9036	19728
Frodo*	79325	79699	79145	26600	51976	72592

Table 4: Comparison Scabbard with NIST finalist KEMs on Cortex-M4 for security level 3
* Collected from pqm4 [KRSS] † Collected from the official website of Saber for high-speed version.

5 Hardware acceleration

The design of new cryptographic schemes prioritizes the security first and foremost. Efficiency also plays an important role in the design decisions, but it is usually considered

in theoretical terms, i.e., algorithmic complexity, which often leads to software efficiency. The short development cycle of software allows fast prototyping and a better feedback loop between developers and designers. However, as we explained in Sec. 1, one of the motivations of this work is to show that hardware efficiency can also be taken into account as part of the design cycle of cryptographic schemes with a successful outcome.

HW/SW vs full HW design strategies: There are two approaches towards hardware acceleration. A HW/SW co-design implements only the most computationally expensive operations on hardware to provide more flexibility and reduce the design cycle at the expense of not achieving the highest performance. A full HW implementation achieves the highest performance but it requires a longer development cycle. The purpose of the hardware implementations is to demonstrate the benefits of our design decisions rather than providing thoroughly optimized processors for the highest performance. Therefore, we choose a HW/SW co-design approach.

If we look at Table 9 in Appendix D, we can observe that the two critical operations in our schemes are polynomial multiplication followed by hashing. As we discussed in Sec. 3 and 4.2, pseudo-random number generation is based on Keccak to make our schemes fairly comparable to the state-of-the-art. At this moment there is a lack of transparency regarding the choice of pseudo-random number generators (PRNG). NIST encourages to use one of the NIST standardized symmetric-key schemes but never specified Keccak. It is quite possible that in the future more efficient constructions may replace the Keccak based PRNG, both in our schemes as well as in NIST finalists. Also, KEMs like Kyber [ABD+21] or Saber [BMD+20] have proposed alternative constructions in their recent NIST submissions using pseudo-random number generators based on AES-CTR named Kyber90s and Saber90s, respectively. Including a Keccak hardware, e.g., the official implementation [Kec], incurs in the same area overhead for any scheme because the functions used are identical. Hence, it does not provide a scientific added value when it comes to comparing different schemes among them. On the other hand, including a Keccak module in hardware would benefit the overall performance of the schemes in the same way that a full hardware implementation outperforms SW/HW co-design approaches.

Since our design decisions were centered around improving the polynomial multiplication in our KEMs, our goal is to demonstrate the improvements in our cryptographic designs using off the shelf and state-of-the-art implementations of polynomial multiplications in hardware. Future works exploring different hardware architectures exploiting more specific properties of each scheme can come later as it has always happened when designing new schemes. In fact, it is very common in the literature to focus on optimizing the polynomial multiplication exclusively when researching on accelerating lattice-based cryptography [ACC+20, CHK+20, KRS18, LS19, MKV20, MTK+20]. Also there is a lot of precedence of outsourcing the most computationally expensive components to hardware accelerators in elliptic-curve or Rivest-Shamir-Adleman cryptography [GFSV09, LXJL11]. Lastly, with our implementations we also demonstrate that different trade-offs between area and performance in hardware implementations can be achieved by exploring the design space of the cryptographic scheme rather than by exploring different hardware architectures.

HW/SW interface: We implement our hardware on a Xilinx Zynq device that integrates FPGA and ARM processors. The communication between them is based on the AXI interface. The commands are transferred in parallel as a single word of 64-bits, that indicates the base address for the memory accesses and the operation performed. The overhead introduced by the commands is negligible since it is only $0.2\mu s$ per command. The data is transferred in a stream free from addressing information and we use the DMA provided by Xilinx to achieve high performance on bulky data transfers. The data transfer of one polynomial (of 256 coefficients), one vector and one matrix takes $2.7\mu s$, $4.4\mu s$ and $14.9\mu s$, respectively. While this overhead is relevant when compared to the multiplication

time in hardware (see Table 6), it can be avoided by having a full hardware implementation. However, as we explained before, our goal is to demonstrate how to achieve efficiency by design rather than providing high performance architectures for given schemes. Also, as we show in Sec. 5.4, our co-processors are effective in accelerating the polynomial arithmetic and set the base around which a full hardware implementation shall be built.

5.1 Florete on hardware

Following a complexity theory analysis, schemes built upon ideal lattices are inherently efficient by design. As we have described in Sec. 3.2, generating an RLWR sample in Florete requires a 768×768 polynomial multiplication, that in turn can be decomposed into 5 256×256 polynomial multiplications applying Toom-Cook 3-way. This means 45% less multiplications than a module lattice-based scheme offering the same security level, e.g., Saber or Kyber, which require 9 256×256 polynomial multiplications for matrix-vector multiplication. However, this comes at the price of a large memory overhead in software. The challenge in hardware is to maintain the benefit in performance over module lattice-based schemes while achieving a comparable area.

The first decision when designing an accelerator for Florete is whether to break down the big 768×768 polynomial multiplication into smaller polynomial multiplications or implement a schoolbook algorithm. If we opt for the former, we only need to implement the Toom-Cook 3-way evaluation and interpolation to wrap up the 256×256 polynomial multiplier, which can be implemented as the existing architectures in the literature. If we opt for the latter, the resulting hardware will be 3 times slower and 3 times bigger than the state-of-the-art 256×256 polynomial multipliers [DFAG19, RB20]. Moreover, since we are following a HW/SW co-design approach rather than an full hardware approach, we can first apply Toom-Cook 3-way on software, and then reuse any 256×256 polynomial multiplier available in the literature to perform the 5 multiplications on hardware. This allows a more fine grain tuning of the implementation since we can trade-off area for speed depending on the needs of our application. Fig. 2 summarizes the data flow and the partition between software and hardware. Instantiating 5 256×256 multipliers in parallel yields to a 5 times larger area while achieving the same performance for a 768×768 multiplication as for a single 256×256 multiplication. Alternatively, we can utilize as little area as for a single 256×256 multiplication to perform the full 768×768 multiplication in 5 times more clock cycles. Since we want to show that the improved efficiency is due to our design of Florete rather than to a carefully optimized implementation, we choose to instantiate only one 256×256 polynomial multiplier.

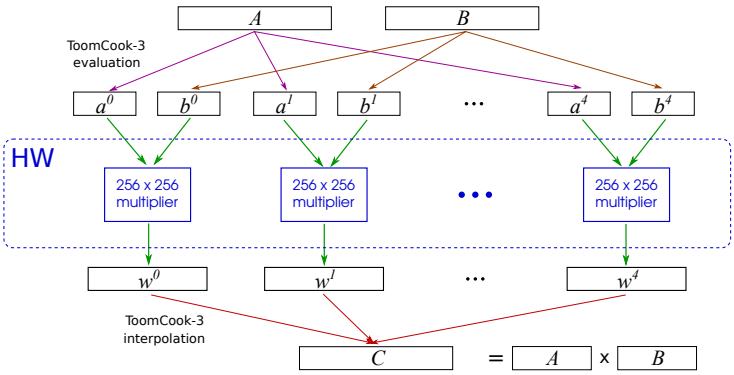

Figure 2: Proposed HW/SW partition for a Florete accelerator

Regarding the choice for the 256×256 polynomial multiplier, we consider the three options available in the literature. The first [DFAG19] implements a schoolbook multiplier

that instantiates 256 multiply-and-accumulate (MAC) units in parallel to perform the innermost loop in a single clock cycle, thus iterating only over the outermost loop. The MAC units are implemented using the DSP primitives available in the FPGA. The problem of this approach is that such a large number of DSP units is only available in the most high end FPGAs, like the UltraScale+ family of Xilinx where it has been implemented. In more standard devices the available number of DSPs imposes a limitation for implementing this approach. The second option [MTK+20] implements a 256×256 multiplier that uses Toom-Cook 4-way to break down one multiplication into 7 64×64 polynomial multiplications that are performed in parallel by compact units. While the performance is worse than the previous method, the design is very compact and it can be applied directly on any FPGA requiring only 28 DSP units, and for any coefficient size for both operands. The third option [RB20] implements a shift register based approach as in the first option, but eliminates the need of DSP units by taking advantage of the fact that the coefficients of one operand, the secret vector, are small. Instead, custom MAC units based on a coefficient-wise shift-and-add are implemented. In addition to this, the latency is halved by embedding the negacyclic convolution in the multiplication. We cannot take advantage of this because the result should be unwrapped for the interpolation of Toom-Cook 3-way. Furthermore, we cannot take advantage of the same shift-and-add MAC units because prior to the 256×256 multiplication the operands grow due to the Toom-Cook 3-way evaluation. Therefore, we implement a design based on [MTK+20]. Results are discussed in Sec. 5.4 and compared to other implementations in our suite and in the state-of-the-art.

5.2 Espada on hardware

Module lattices are in general friendly towards parallelization due to the matrix-vector multiplication. Furthermore, when we use an algorithm to break a large polynomial multiplication into several smaller polynomial multiplications, e.g., using Toom-Cook, we are generating new parallelisms. However, the latter comes with a computational overhead due to the evaluation and interpolation steps of such algorithm. The parameter choices for Espada seek to exploit the inherent parallelism of matrix-vector multiplication even further while avoiding the extra cost of breaking down large polynomial multiplications. This translates into compactness by design in single instruction single data (SISD) processors as shown in Sec. 4.2. The drawback of Espada with respect to other lattice-based KEMs is the increased number of polynomial multiplications and randomness requirements. The challenge in hardware is to exploit the parallelism effectively to bring Espada performance close to the state-of-the-art.

Fig. 3 shows our proposed architecture to exploit the parallelism of Espada's matrix-vector multiplication. For a public matrix of dimension $l \times l$, l polynomial multipliers are instantiated in parallel. Each of the parallel multipliers is fed with a row of the public matrix. The second operand. which is the corresponding polynomial of the secret on each iteration, is the same for all multipliers. Each multiplier reads and writes data to a small local memory implemented as LUT-based RAM to avoid the large penalization of accessing in parallel the system memory. This distributed memory is also filled at the same time as the polynomials are being transferred to the memory to minimize the loading penalization. The second operand can be sent simultaneously to all multipliers, so it does not incur an additional overhead. The result accumulated on the i-th multiplier corresponds to the i-th row-vector product. Note that this architecture exploits the parallelism at matrix-vector level while still leaves room for a certain trade-off between area and performance with the design of the 64×64 polynomial multipliers.

The proposed parameters for achieving a NIST security level 3 with Espada is set as $l = 12$. While allowing a high degree of parallelization, this also imposes an important constraint on the design of the 64×64 polynomial multipliers. Figure 4 shows a parameterizable architecture for such multiplier. The number of arithmetic units, implemented with

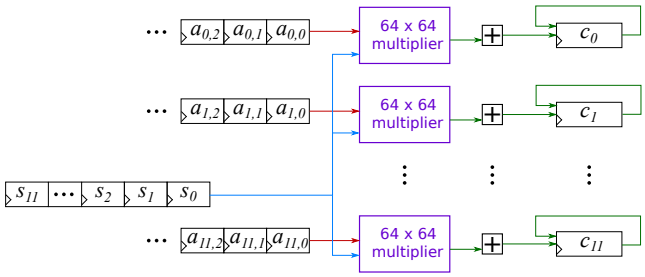

Figure 3: Parallel architecture for matrix-vector multiplication in Espada

native DSP primitives for efficiency, can be increased or decreased for achieving a higher performance or a lower area utilization. In our implementation, this circuit will be instantiated 12 times in parallel. To guarantee that the FPGA resources will not be exhausted, we choose 4 DSP units per polynomial multiplier, which add up to 48 overall. In Sec. 5.4 we include performance and area results and discuss them in detail.

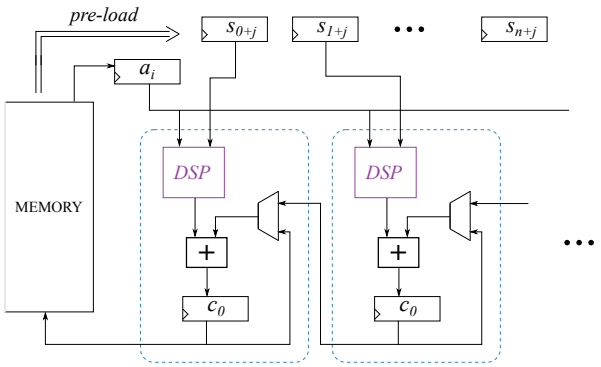

Figure 4: Architecture of the compact 64×64 polynomial multiplier used for Espada

5.3 Sable on hardware

In contrast with Florete and Espada that are not similar to existing schemes, the design of Sable is close to Saber. As explained in Sec. 3.4, our contribution here is on the selection of more efficient parameters for Saber by applying the latest results of research on lattices. From an implementation point of view, all the existing research on Saber implementations can be applied directly to Sable. In particular, the implementations in [DFAG19] and [MTK+20] should be readily available to support Sable parameters owing to the higher flexibility offered by HW/SW co-designed accelerators. The processor in [RB20] implements an instruction set architecture (ISA) with a unified sampling module to give support to LightSaber, Saber and FireSaber parameters. These three parameter sets sample the secrets from a centered binomial distribution with different η values, being $\eta = 5, 4, 3$, respectively. This module being extended to support also $\eta = 1$, such processor shall also serve for accelerating Sable. Moreover, any arithmetic module of a Saber processor can be directly reused for Sable after sign-extending the most significant bit of every secret coefficient.

Although existing Saber co-processors can be reused for Sable, the secrets of our scheme are smaller. The architectures in [DFAG19] and [MTK+20] are more generic, but the one proposed in [RB20] exploits the small secrets to achieve high performance without exhausting the FPGA resources. We optimize their architecture for our parameters, which

allows us to substantially reduce the area requirements without a performance loss. Figure 5 draws our proposed architecture. The dashed box highlights the arithmetic unit, which is instantiated 256 times for a full parallel multiplication. For the arithmetic unit we use a custom architecture which is very efficient for 2 bits secrets. If the least significant bit of the secret coefficient is zero, the value in the accumulator register does not change, and it is irrelevant if the other secret bit is 0 or 1. If the least significant bit of the secret is one, the current coefficient of a will be either add or subtract from the result depending on the most significant bit of the secret. The multiplier implements 256 arithmetic units for a fully parallelized multiplication. Our parameters allow us to pack more secret coefficients in less memory, therefore we can reduce the overhead for loading the secret register. The negacyclic convolution is performed in-place and the result is stored in the accumulator registers. These registers can be reset to perform a polynomial multiplication, or preserved to perform the row-column multiplication in the matrix-vector multiplication saving up the time spent on the additions. The performance and area figures of our design are discussed in Sec. 5.4 as for the implementations of Florete and Espada.

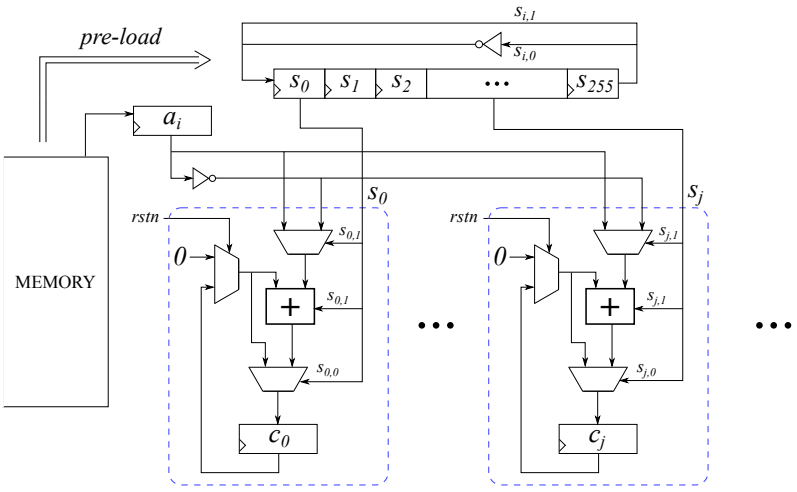

Figure 5: Parallel architecture of Sable polynomial multiplier

5.4 Results

We have implemented all our hardware designs using the Vivado Design Suite 2018.1 and targeting the Xilinx ZedBoard Zynq-7000 AP SoC XC7Z020-CLG484 and runnning the synthesis and implementation with the default strategies. We use the ARM Cortex-A9 CPU running at 666 MHz available in the same chip to send the data and commands to the hardware accelerator, which in turn runs at 125 MHz for Florete and Espada, and at 150 MHz for Sable. Table 5 shows a comparison for each scheme in Scabbard when implemented only on SW or accelerated by the multiplier available in hardware. For Espada we use the multiplier described in Sec. 5.2, while for Sable we show the speed-ups that can be obtained by the compact multiplier described in Sec. 5.1 and by the high speed multiplier described in Sec. 5.3. We include Saber because the implementation in [MTK+20] also follows a HW/SW co-design approach which allows a fair comparison. We include the total area requirements of these three HW/SW co-processors in Appendix F. In the following, we restrict our discussion to the multiplier architectures, which is the only block implemented on hardware, and allows us a fair comparison to the state of the art.

In Table 6 we show the area requirements of the polynomial multiplier and compare it with the compact multiplier for Saber in [MTK+20] and the high performance multiplier

Scheme name	Performance on Zedboard (SW only) - [μs]			Performance on Zedboard (HW/SW) - [μs]			Speed up		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Espada	51373	55706	58527	9618	10460	9820	5.3	5.3	6.0
Sable*	47032	62611	77095	2406	3114	2765	19.5	20.1	27.9
Sable†				2958	3714	3419	15.9	16.9	22.5
Saber [MTK+20]	17659	22438	27001	3273	4147	3844	5.4	5.4	7.0

Table 5: Performance comparison between schoolbook implementations on software and the speed-up achieved by the polynomial multiplier on hardware.
* using the high speed 256×256 multiplier † using the compact 256×256 multiplier

for Saber in [RB20]. In particular, we compare the compact designs for Florete and Espada in rows 1 and 2 with the compact Saber processor from [MTK+20] in row 4, and the high speed multiplier for Sable in row 3 to the fast Saber multiplier from [RB20] in row 5. We choose to compare our designs with Saber because it is the most well-known LWR-based scheme and all our schemes are also based on variants of the LWR problem. Also, note that we compare the performance of a full matrix-vector multiplication rather than a single polynomial multiplication because otherwise Florete may seem the least efficient solution when it is actually the opposite, and Espada may seem an order of magnitude faster due to the small polynomials and large matrix dimension.

	Scheme	Platform	f	LUTs	FFs	DSPs	t_{mvmul}
1	Florete [Ours]	Zedboard	125	2,878	1,263	38	88 μs
2	Espada [Ours]	Zedboard	125	4,150	2,537	48	111 μs
3	Sable [Ours]	Zedboard	150	6,084	3,354	0	17 μs
4	Saber [MTK+20]	Zedboard	125	2,927	1,279	38	158 μs
5	Saber [RB20]	Ultrascale+	250	17,429	5,083	0	11 μs

Table 6: Area and performance results of the polynomial arithmetic on hardware for our schemes and state-of-the-art Saber

Florete and Espada follow two different approaches for improving the efficiency of a KEM with respect to Saber. Florete is a ring scheme, which means that it is inherently faster in software, as shown in Table 3 and 4. In hardware this translates into achieving a higher performance than Saber when using equivalent architectures. We can observe this by comparing the results of the first and fourth rows in Table 6. The small ring chosen to build the module-lattice problem in Espada makes it inherently compact on software, as shown in Table 4. Despite being considerably slower on software than other schemes, this difference can be mitigated, or even overcome for the comparison with Saber, thanks to the highly parallelizable matrix-vector multiplication. In this work we have opted for a compact design of the parallel multipliers which has turn in a reduced area consumption that still outperforms Saber. We leave as future work the exploration of high performance architectures for Florete and Espada. Lastly, in Sable we have exploited the improved parameters to reduce the size of the secrets. To demonstrate the success of our approach, we have implemented a multiplier architecture that exploits this fact, similarly to the implementation of Saber in the last row. Comparing the third and fifth rows, we can observe that our approach greatly reduces the area requirements. As for the performance, it should be noted that the clock cycles of both designs are nearly equivalent, but the superior technology of UltraScale+ boards with respect to Zedboards allows a higher operating frequency which translates into a faster execution time.

6 Conclusion

We have provided a suite of lattice-based KEMs which improves upon almost all of the practical aspects of state-of-the-art. We alluded many research directions throughout this work and our techniques can be readily adapted for different schemes. Although we provide optimal implementations of our schemes and suggest architectures for hardware acceleration, we strongly believe that more research is necessary on the implementation aspects. We also plan to provide different parameter sets to satisfy different security requirements in the future. In conclusion, we believe this work will open up a new research direction and it will inspire more people to work further in this direction.

7 Acknowledgements

This work was supported partially by the Research Council KU Leuven: C16/15/058, by the European Commission through the Horizon 2020 research and innovation programme under grant agreement Cathedral ERC Advanced Grant 695305 and grant agreement H2020-DS-LEIT-2017-780108 FENTEC, and by CyberSecurity Research Flanders with reference number VR20192203. In addition, Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as junior post-doctorate fellow.

References

[AAJB+19] Erdem Alkim, Roberto Avanzi, Léo Ducas Joppe Bos, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. Newhope : Algorithm specifications and supporting documentation. Second PQC Standardization Conference, 2019, University of California, Santa Barbara, USA, 2019.

[AASA+17] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moodyand Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the second round of the nist post-quantum cryptography standardization process. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf, 2017. [Online; accessed 10-Oct-2020].

[ABC19] Erdem Alkım, Yusuf Alper Bilgin, and Murat Cenk. Compact and Simple RLWE Based Key Encapsulation Mechanism. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 237–256, Cham, 2019. Springer International Publishing.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancréde Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-kyber. algorithm specifications and supporting documentation. (round 2 submission). https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf, 2019. [Online; accessed 30-January-2021].

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancréde Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber. algorithm specifications and supporting documentation. (round 3 submission). https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf, 2021. [Online; accessed 30-January-2021].

[ACC+20] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial multiplication in ntru prime: Comparison of optimization strategies on cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, Dec. 2020.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 595–618, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[ADP18] Martin Albrecht, Amit Deo, and Kenneth Paterson. Cold boot attacks on ring and module lwe keys under the NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):173–213, Aug. 2018.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *USENIX Security 2016*, 2016.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046.

[Bar87] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[BBG+17] Hayo Baan, Sauvik Bhattacharya, Óscar García-Morchón, Ronald Rietman, Ludo Tolhuizen, Jose Luis Torre-Arce, and Zhenfei Zhang. Round2: KEM and PKE based on GLWR. *IACR Cryptol. ePrint Arch.*, 2017:1183, 2017.

[BCLvV16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: reducing attack surface at low cost. Cryptology ePrint Archive, Report 2016/461, 2016. https://eprint.iacr.org/2016/461.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[BDK+20] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733, 2020. https://eprint.iacr.org/2020/733.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. Online, 2001. https://cr.yp.to/papers/m3.ps.

[BGML+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: KEM and PKE based on GLWR. Cryptology ePrint Archive, Report 2018/725, 2018. https://eprint.iacr.org/2018/725.

[BMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 3 Submission). https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf, 2020. [Online; accessed 30-January-2021].

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. Cryptology ePrint Archive, Report 2018/355, 2018. https://eprint.iacr.org/2018/355.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT 2012*, pages 719–737, 2012.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols (extended version). Cryptology ePrint Archive, Report 2019/1140, 2019. https://eprint.iacr.org/2019/1140.

[BZ06] Marco Bodrato and Alberto Zanoni. What about Toom-Cook Matrices Optimality. Author's website, 2006. http://marco.bodrato.it/papers/WhatAboutToomCookMatricesOptimality.pdf.

[CCLS20] Hao Chen, Lynn Chua, Kristin Lauter, and Yongsoo Song. On the concrete security of lwe with small secret. Cryptology ePrint Archive, Report 2020/539, 2020. https://eprint.iacr.org/2020/539.

[CDH+19] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU algorithm specifications and supporting documentation,. Second PQC Standardization Conference, 2019, University of California, Santa Barbara, USA, 2019.

[CHK+20] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. Ntt multiplication for ntt-unfriendly rings. Cryptology ePrint Archive, Report 2020/1397, 2020. https://eprint.iacr.org/2020/1397.

[CKLS18] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. Lizard: Cut off the tail! A practical post-quantum public-key encryption from LWE and LWR. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, volume 11035 of *Lecture Notes in Computer Science*, pages 160–177. Springer, 2018.

[CN11] Yuanmi Chen and Phong Q. Nguyen. Bkz 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Coo66] S. A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966. pp. 51-77.

[DDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 329–358. Springer, 2020.

[DFAG19] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 206–214, 2019.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[Din12] Jintai Ding. A simple provably secure key exchange scheme based on the learning with errors problem. *IACR Cryptol. ePrint Arch.*, 2012:688, 2012.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, 2018.

[DKRV19] Jan Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Mod-LWR based kem,. Second PQC Standardization Conference, 2019, University of California, Santa Barbara, USA, 2019.

[DKSRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-Secure Encryption and CCA-Secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2018*, pages 282–305, Cham, 2018. Springer International Publishing.

[DSDGR20] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. LWE with side information: Attacks and concrete security estimation. Cryptology ePrint Archive, Report 2020/292, 2020. https://eprint.iacr.org/2020/292.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. Cryptology ePrint Archive, Report 2019/292, 2019. https://eprint.iacr.org/2019/292.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, 2018. [Online; accessed 10-October-2020].

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[GFSV09] Xu Guo, Junfeng Fan, Patrick Schaumont, and Ingrid Verbauwhede. Programmable and parallel ecc coprocessor architecture: Tradeoffs between area, speed and security. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 289–303, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GJY19] Qian Guo, Thomas Johansson, and Jing Yang. A novel CCA attack using Decryption Errors against LAC. Cryptology ePrint Archive, Report 2019/1308, 2019. https://eprint.iacr.org/2019/1308.

[GMR20] Aurelien Greuet, Simon Montoya, and Guenael Renault. Attack on LAC key exchange in misuse situation. Cryptology ePrint Archive, Report 2020/063, 2020. https://eprint.iacr.org/2020/063.

[Goo52] I.J. Good. Random motion on a finite abelian group. *Mathematical Proceedings of the Cambridge Philosophical Society*, 48(2):368–368, 1952.

[GPV07] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. Cryptology ePrint Archive, Report 2007/432, 2007. https://eprint.iacr.org/2007/432.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):372–393, 2018.

[JZC+17] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. Post-quantum IND-CCA-secure KEM without additional hash. *IACR Cryptol. ePrint Arch.*, 2017:1096, 2017.

[KBMSRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on arm. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, Aug. 2018.

[Kec] Keccak Team. Hardware resources - Keccak in VHDL. [Online; accessed 08-February-2021].

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of USSR Academy of Sciences*, 145(7):293–294, 1962.

[KRS18] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Cryptology ePrint Archive, Report 2018/1018, 2018. https://eprint.iacr.org/2018/1018,.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4; Accessed 30-January-2021.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, and Kunpeng Wang. LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. Cryptology ePrint Archive, Report 2018/1009, 2018. https://eprint.iacr.org/2018/1009.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, Jun 2015.

[LS19] Vadim Lyubashevsky and Gregor Seiler. Nttru: Truly fast ntru using ntt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, May 2019.

[LXJL11] Z. Liu, L. Xia, J. Jing, and P. Liu. A tiny rsa coprocessor based on optimized systolic montgomery architecture. In *Proceedings of the International Conference on Security and Cryptography*, pages 105–113, 2011.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Cryptol. ePrint Arch.*, 2020:268, 2020.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 44(2):519–521, 1985.

[MTK+20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism. *IACR Cryptol. ePrint Arch.*, 2020:321, 2020.

[NIS17] NIST. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, 2017. [Online; accessed 10-Oct-2020].

[Pei14] Chris Peikert. Lattice cryptography for the internet. *IACR Cryptol. ePrint Arch.*, 2014:70, 2014.

[Pol71] J. M. Pollard. The fast fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. Cryptology ePrint Archive, Report 2017/594, 2017. https://eprint.iacr.org/2017/594.

[RB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Cryptol. ePrint Arch.*, 2020:434, 2020.

[Reg04] Oded Regev. *New Lattice-based Cryptographic Constructions*, volume 51-6, pages 899–942. ACM, New York, NY, USA, November 2004.

[RSSS17] Miruna Roşca, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Middle-product learning with errors. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 283–297, Cham, 2017. Springer International Publishing.

[SC19] Yongha Son and Jung Hee Cheon. Revisiting the hybrid attack on sparse and ternary secret lwe. Cryptology ePrint Archive, Report 2019/1019, 2019. https://eprint.iacr.org/2019/1019.

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[Son19] Yongha Son. A note on parameter choices of round5. Cryptology ePrint Archive, Report 2019/949, 2019. https://eprint.iacr.org/2019/949.

[SSZ19] Ron Steinfeld, Amin Sakzad, and Raymond K. Zhao. Practical MP-LWE-based encryption balancing security-risk vs. efficiency. Cryptology ePrint Archive, Report 2019/1179, 2019. https://eprint.iacr.org/2019/1179.

[Too63] A.L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics-Doklady*, volume 7, pages 714–716, 1963. http://toomandre.com/my-articles/engmat/MULT-E.PDF.

[TRG17] Ludo Tolhuizen, Ronald Rietman, and Óscar García-Morchón. Improved key-reconciliation method. *IACR Cryptol. ePrint Arch.*, 2017:295, 2017.

A IND-RND security of LWR based KEX

The advantage of the adversary $\mathscr{A}$ in distingishing between an output from a pseudo-random generator (prg) f and a random sample from uniform distribution is [DKSRV18] defined by

$$\mathrm{Adv}_f^{\mathrm{prg}}(\mathscr{A}) = \left| \begin{array}{c} Pr\left[b = 1 : \begin{array}{c} r \leftarrow \mathscr{U}(\{0,1\}^m); \\ s \leftarrow f(r) \in (\{0,1\}^n); \\ b = \mathscr{A}(s); \end{array}\right] \\ \\ -Pr\left[b = 1 : \begin{array}{c} s \leftarrow \mathscr{U}(\{0,1\}^n) \\ b = \mathscr{A}(s); \end{array}\right] \end{array} \right|,$$

where $n_1 \leq n_2$. Then f is called a pseudo-random generator if for any adversary $\mathscr{A}$, $\mathrm{Adv}_f^{\mathrm{prg}}(\mathscr{A}) \leq negligible$.

The advantage of an adversary $\mathscr{A}$ against the LWR problem is defined by

$$\mathrm{Adv}^{\mathrm{LWR}}(\mathscr{A}) = \left| \begin{array}{c} Pr\left(b' = 1 : \begin{array}{c} \mathbf{A} \leftarrow \mathscr{U}((\mathscr{R}_q^n)^{l\times l});\ \mathbf{s} \leftarrow \beta_\eta((\mathscr{R}_q^n)^l); \\ b' = \mathscr{A}(\mathbf{A}, \lfloor (p/q)\mathbf{A}\cdot\mathbf{s}\rceil); \end{array}\right) \\ \\ -Pr\left(b' = 1 : \begin{array}{c} \mathbf{A} \leftarrow \mathscr{U}((\mathscr{R}_q^n)^{l\times l});\ \mathbf{s} \leftarrow \beta_\eta((\mathscr{R}_q^n)^l); \mathbf{b} \leftarrow \mathscr{U}((\mathscr{R}_p^n)^l); \\ b' = \mathscr{A}(\mathbf{A}, \mathbf{b}); \end{array}\right) \end{array} \right|.$$

As LWR problem is hard, $\mathrm{Adv}^{\mathrm{LWR}}(\mathscr{A})$ is negligible.

Theorem 4. *LWR based KEX is IND-RND secure if $q/p \leq p/(2^B t)$.*

Proof. For proving (Ring/Module)LWR based KEX is IND-RND secure, we need to show that for every adversary $\mathscr{A}$

$$\mathrm{Adv}_{\mathrm{KEX}}^{\text{ind-rnd}}(\mathscr{A}) \leq negligible\,.$$

It is sufficient to show that, for any adversary $\mathscr{A}$ against the KEX, there exists three adversaries $\mathscr{B}_1, \mathscr{B}_2, \mathscr{B}_3$ such that

$$\mathrm{Adv}_{\mathrm{KEX}}^{\text{ind-rnd}}(\mathscr{A}) \leq \mathrm{Adv}_{Gen}^{\mathrm{prg}}(\mathscr{B}_1) + \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_2) + \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_3)\,.$$

To prove that, we define six indistinguishability games G_i where $i = 1, 2, \cdots, 6$ in Fig. 6. Let $E_i^{\mathscr{A}}$ be the event when adversary $\mathscr{A}$ wins the game G_i. The advantage of any adversary $\mathscr{A}$ against the game G_i is defined by $\mathrm{Adv}_{G_i}(\mathscr{A}) = \left|Pr\left[E_i^{\mathscr{A}}\right] - 1/2\right|$. Note that, $\mathrm{Adv}_{G_1}(\mathscr{A}) = \mathrm{Adv}_{\mathrm{KEX}}^{\text{ind-rnd}}(\mathscr{A})$. In game G_2, the public matrix(in case of ring the dimension of the matrix is 1×1) is generated randomly, not by the pseudo-random generator $\mathtt{gen}_n^{l\times l}()$. If any adversary $\mathscr{B}_1$ can differentiate between these two games, then it can distinguish between the pseudo-randomly generated polynomial and the randomly generated polynomial. Hence, $\left|Pr\left[E_1^{\mathscr{A}}\right] - Pr\left[E_2^{\mathscr{A}}\right]\right| \leq \mathrm{Adv}^{prg}(\mathscr{B}_1)$. In game G_3, $\mathbf{b}$ is generated uniformly random from $(\mathscr{R}_p^n)^l$ and $(\mathbf{A}, \mathbf{b})$ is a uniformly generated random sample. But, in game G_2, $(\mathbf{A}, \mathbf{b})$ is sampled from a LWR distribution. If any adversary $\mathscr{B}_2$ can distinguish between these two games, then it can solve the decisional LWR problem. Therefore, $\left|Pr\left[E_2^{\mathscr{A}}\right] - Pr\left[E_3^{\mathscr{A}}\right]\right| \leq \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_2)$. In game G_3, c' is reduced by $\epsilon_p - \epsilon_t - B$ bits, whereas in Game G_4, this number is $\epsilon_q - \epsilon_p$. As $(\epsilon_q - \epsilon_p) \leq (\epsilon_p - \epsilon_t - B)$ then in game G_4, we are losing fewer or equal bits. Then the advantage is greater or equal in game G_4. Hence, for any adversary $\mathscr{A}$ there exists another adversary $\mathscr{A}'$ such that $\mathrm{Adv}_{G_3}(\mathscr{A}) \leq \mathrm{Adv}_{G_4}(\mathscr{A}')$. Then, $\left|Pr\left[E_3^{\mathscr{A}}\right] - Pr\left[E_3^{\mathscr{A}'}\right]\right| \leq 0$. In game G_5, c' is generated from the multiplication of $\mathbf{b}$ with l entries from $\mathscr{R}_q^n$ and $\mathbf{s}'$ instead of $bits(\mathbf{s}', \epsilon_p, \epsilon_p)$. It will not change the result c'. As $p|q$, then generating the polynomial $\mathbf{b}$ from $\mathscr{U}^l(\mathscr{R}_q^n)$ in place of $\mathscr{U}^l(\mathscr{R}_p^n)$ gives more or at least same advantage to the adversary of game G_5 than

Game G_1:

1. $seed_{\mathbf{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2. $\mathbf{A} \leftarrow \mathtt{gen}_n^{l\times l}(\mathrm{seed}_{\mathbf{A}})$
3. $\mathbf{s}, \mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
4. $\mathbf{b} = \mathtt{bits}(\mathbf{A}\cdot\mathbf{s} + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
5. $\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
6. $u' = \mathbf{b}^T \cdot \mathtt{bits}(\mathbf{s}', \epsilon_p, \epsilon_p) + h_1$
7. $c' = \mathtt{bits}(u', \epsilon_p - B, \epsilon_t)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Game G_2:

1.
2. $\mathbf{A} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^{l\times l})$
3. $\mathbf{s}, \mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
4. $\mathbf{b} = \mathtt{bits}(\mathbf{A}\cdot\mathbf{s} + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
5. $\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
6. $u' = \mathbf{b}^T \cdot \mathtt{bits}(\mathbf{s}', \epsilon_p, \epsilon_p) + h_1$
7. $c' = \mathtt{bits}(u', \epsilon_p - B, \epsilon_t)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Game G_3:

1.
2. $\mathbf{A} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^{l\times l})$
3. $\mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
4. $\mathbf{b} \leftarrow \mathcal{U}((\mathscr{R}_p^n)^l)$
5. $\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
6. $u' = \mathbf{b}^T \cdot \mathtt{bits}(\mathbf{s}', \epsilon_p, \epsilon_p) + h_1$
7. $c' = \mathtt{bits}(u', \epsilon_p - B, \epsilon_t)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Game G_4:

1.
2. $\mathbf{A} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^{l\times l})$
3. $\mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
4. $\mathbf{b} \leftarrow \mathcal{U}((\mathscr{R}_p^n)^l)$
5. $\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
6. $u' = \mathbf{b}^T \cdot \mathtt{bits}(\mathbf{s}', \epsilon_p, \epsilon_p) + h_1$
7. $c' = \mathtt{bits}(u', \epsilon_q - B, \epsilon_p - B)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Game G_5:

1.
2. $\mathbf{A} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^{l\times l})$
3. $\mathbf{s}' \leftarrow \beta_\eta((\mathscr{R}_q^n)^l)$
4. $\mathbf{b} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^l)$
5. $\mathbf{b}' = \mathtt{bits}(\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h_1}, \epsilon_q, \epsilon_p)$
6. $u' = \mathtt{bits}(b^T \cdot s' + h_1, \epsilon_q, \epsilon_p)$
7. $c' = \mathtt{bits}(u', \epsilon_p - B, \epsilon_p - B)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Game G_6:

1.
2. $\mathbf{A} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^{l\times l})$
4. $\mathbf{b} \leftarrow \mathcal{U}((\mathscr{R}_q^n)^l)$
5. $\mathbf{b}' \leftarrow \mathcal{U}((\mathscr{R}_p^n)^l)$
6. $u' \leftarrow \mathcal{U}(\mathscr{R}_p^n)$
7. $c' = \mathtt{bits}(u', \epsilon_p - B, \epsilon_p - B)$
8. $k' = \mathtt{bits}(u', \epsilon_p, B)$
9. $\hat{k} \leftarrow \mathcal{U}(\mathscr{R}_{2^B}^n)$
10. $u \leftarrow \mathcal{U}(\{0,1\})$
11. if $u = 0$: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', k')$
12. else: return$(\mathbf{A}, \mathbf{b}, \mathbf{b}', c', \hat{k})$

Figure 6: Sequence of games that are used in the proof of Theorem 4

the adversary of G_4. Eliminating the last $\epsilon_q - \epsilon_p$ bits of u' will not affect this as they are not needed further. For any adversary $\mathscr{A}'$ there exists another adversary $\mathscr{A}''$ such that $\mathrm{Adv}_{G_4}(\mathscr{A}') \leq \mathrm{Adv}_{G_5}(\mathscr{A}'')$. Then, $\left|Pr\left[E_4^{\mathscr{A}'}\right] - Pr\left[E_5^{\mathscr{A}''}\right]\right| \leq 0$. In game G_5, $(\mathbf{b}, u')$ is a LWR problem, but in G_6, these are sampled uniformly random from $(\mathscr{R}_q^n)^l$ and $\mathscr{R}_p^n$ respectively. If there is an adversary $\mathscr{B}_3$ who can distinguish these two games, it can solve the decisional LWR problem. Consequently, $\left|Pr\left[E_5^{\mathscr{A}}\right] - Pr\left[E_6^{\mathscr{A}}\right]\right| \leq \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_3)$. In game G_6, b, b', u' all are independently and uniformly sampled. Since k' is the first B

bits of u', then it is also uniformly distributed. Hence, $\left|Pr\left[E_6^{\mathscr{A}''}\right]\right| = 1/2$. Combining all of these, we get

$$\mathrm{Adv}_{G_1}(\mathscr{A}) = \left|Pr\left[E_1^{\mathscr{A}}\right] - 1/2\right| \leq \mathrm{Adv}^{\mathrm{prg}}_{Gen}(\mathscr{B}_1) + \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_2) + \mathrm{Adv}^{\mathrm{LWR}}(\mathscr{B}_3)\,.$$

□

B Bound in the error tolerance of the error correction scheme

Below we establish the bound of error tolerance of the error correction scheme shown by [Pei14, Din12]. Our proof is a simpler alternative of the proof given by Tolhuizen *et al.* [TRG17].

Theorem 2. If $x = y + e$ and $|e| \leq \frac{q}{2^{B+1}} - \frac{q}{2^{B+\epsilon_t+1}}$, then

$$\texttt{Encode}(x) = \texttt{Encode}(\texttt{Decode}(y, \texttt{HelpDecode}(x))).$$

Proof. We can write x as $x = s \cdot \frac{q}{2^B} + h \cdot \frac{q}{2^{B+\epsilon_t}} + g$, where $0 \leq s < 2^B$, $0 \leq h < \frac{q}{2^B}$ and $0 \leq g < \frac{q}{2^{B+\epsilon_t}}$. Here, $\texttt{HelpDecode}(x) = h$. Then, $\texttt{Decode}(y, h) = y - h\frac{q}{2^{B+\epsilon_t}} = x - e - h\frac{q}{2^{B+\epsilon_t}} = s \cdot \frac{q}{2^B} + g - e$.

$$\begin{aligned}\text{Now, } |g - e| &\leq \frac{q}{2^{B+\epsilon_t}} - 1 + \frac{q}{2^{B+1}} - \frac{q}{2^{B+\epsilon_t+1}} = \frac{q}{2^{B+1}} + \frac{q}{2^{B+\epsilon_t+1}} - 1 \\ &\leq \frac{q}{2^{B+1}} + \frac{q}{2^{B+1}} - 1 \text{ as } (\epsilon_t \geq 1) = \frac{q}{2^B} - 1\end{aligned}$$

Hence, $|g - e| \leq \frac{q}{2^B} - 1$, i.e., $|g - e| < \frac{q}{2^B}$. So, $\texttt{Encode}(\texttt{Decode}(y, h)) = s = \texttt{Encode}(x)$.

□

C Extension for NTT-friendly prime fields

A natural extension of our work here is to explore possibilities of instantiating different key-encapsulation mechanisms for NTT-friendly prime fields. Such schemes can be considered analogous to ring- or Module-LWE based schemes such as NewHope [AAJB+19] and Kyber [BDK+17] which have been designed by keeping NTT-friendliness in mind. Also, there is a concurrent work by Chung *et al.* [CHK+20] showed how NTT multiplication can be used to speed-up schemes that uses NTT unfriendly power-of-two fields which are used in this paper. The central idea is choosing a prime p such that it is able to *contain* the largest possible number occurring during the execution of the scheme. For example, in Saber [DKRV19] if the field elements in $\mathbb{Z}_q$ are represented as $[-\frac{q}{2}, \frac{q}{2})$ then the largest possible number can occur during polynomial multiplication in the ring $\mathscr{R}_q^n$ and the absolute magnitude can be at most $nq^2/4$. Hence, if the prime is chosen such that it satisfies $p > nq^2/2$ and $n|(p-1)$ and the multiplication is computed in $\mathscr{R}_p^n$ then the correct result in $\mathscr{R}_q^n$ can be recovered easily due to the choice of p. Additionally, If we consider that one of the multiplicand is sampled from a centered binomial distribution β_η and can have values between $[-\eta, \eta]$ instead of much larger $[-\frac{q}{2}, \frac{q}{2})$ then a smaller prime can be chosen. We call this method embedding technique. In this section, we discuss how the schemes presented in this work can be adapted or modified for NTT-friendly prime fields.

C.1 Florete

A straightforward approach to extend Florete using embedding strategy is to choose a prime p which satisfies the inequalities above and facilitates 256×256 polynomial multiplication

using NTT. Therefore multiplying two polynomials $a, b \in \mathscr{R}_p^{768} = \mathbb{Z}_p/(x^{768} - x^{384} + 1)$ is possible by using a Toom-Cook 3-way evaluation to split each of a, b into 5 polynomials of length 256, multiplying them by NTT, and finally combining the results using Toom-Cook 3-way interpolation. This is followed by a final reduction by the polynomial $x^{768} - x^{384} + 1$. However, this straightforward combination of Toom-Cook 3-way and NTT has many problems which is unlikely to lead to an efficient multiplication strategy. First, we cannot use the *negacyclic-NTT* in this strategy. As the Toom-Cook 3-way interpolation requires the results of 256×256 polynomial multiplication of to be of length 511, i.e., without the final reduction by $x^{256} + 1$. This requires the polynomials after Toom-Cook 3-way evaluation stage to be zero-padded to double their lengths and passed to the NTT routines. Due to this the number of memory accesses within the NTT transformations are more and also requires storing double the amount of twiddle factors. Second, unlike Floret which is much faster than Saber with Toom-Cook and Karatsuba based polynomial multiplication. Embedded version of Floret will be slower than embedded version of Saber. In Saber, to perform a matrix-vector multiplication we need $9 + 3$ forward NTT transformations for the public matrix and the secret. This is followed by 3 inverse-NTT transformations. Whereas in Floret to multiply two polynomial a, b we need forward 10 forward NTT transformations and 5 inverse-NTT transformation. Although the number of NTT transformations here look similar we have to remember that the NTT transformations for Saber is faster than embedded version of Floret. Moreover, in the overall scheme Saber can save the forward NTT transformations of the secrets by computing them once and saving them in NTT formats with little increase in memory requirement. In embedded version of Floret this cannot be done by without increasing the memory usage by a large amount. Finally, the the Toom-Cook 3-way evaluation and interpolation is costlier for embedded version of Floret as the modular reduction is costlier in this scenario. Albeit, this overhead can be reduced by using some techniques such as converting to Montgomery domain [Mon85] or Barrett divisions [Bar87], this overhead is significant when compared to the *free* modular reduction offered by power-of-two moduli. Hence, to apply the embedding method on Florete, we don't think the straightforward method as described above is very suitable.

To apply the embedding method we can apply other better strategies such as i) Good's FFT trick [Goo52, ACC+20, CHK+20] and ii) incomplete-NTT [ABC19, LS19]. The Good's trick although a better choice than the straightforward embedding technique, it requires length doubling of the multiplicand polynomials. Also this is more useful for schemes which uses non-cyclotomic rings such as NTRU [CDH+19].

For schemes like Florete which uses cyclotomic rings we found that the second method most optimal in our analysis. The main idea of this method is that if ζ_1 and ζ_2 are two primitive sixth roots of unity in the underlying field then as $x^2 - x + 1$ is the sixth cyclotomic polynomial we have $\zeta_1 + \zeta_2 = 1$ and $\zeta_1 \cdot \zeta_2 = 1$. Using ζ_1 and ζ_2 we can form the following CRT (Chinese Remainder theorem) map for our ring,

$$\mathbb{Z}_p[x]/(x^{768} - x^{384} + 1) = \mathbb{Z}_p[x]/(x^{384} - \zeta_1) \times \mathbb{Z}_p[x]/(x^{384} - \zeta_2)$$

After this step, we can perform 7 layers of standard NTT on each of $\mathbb{Z}_p[x]/(x^{384} - \zeta_1)$ and $\mathbb{Z}_p[x]/(x^{384} - \zeta_2)$. After this, we are left with 256 3×3 schoolbook multiplication modulo $x^3 \pm 1$ and the reverse NTT steps to complete the multiplication in $\mathbb{Z}_p[x]/(x^{768} - x^{384} + 1)$. Another otpimization is that during the first layer CRT map once we calculate the CRT map modulo $x^{384} - \zeta_1$ we do not need to calculate the CRT map modulo $x^{384} - \zeta_2$ again. Instead we can utilize the fact that $\zeta_1 + \zeta_2 = 1$ and calculate modulo $x^{384} - 1 + \zeta_1$. This saves almost half of the modular multiplication with ζ_2. For this technique we need the prime modulus p such that there exists a γ and $\gamma^{128} = \zeta_1$. As $\zeta_1^6 = 1$, this implies $\gamma^{768} = 1$. Hence additional to satisfying embedding criteria the prime p should satisfy $768|(p-1)$. Using the analysis same as Chung *et al.* [CHK+20] the smallest such p that can be used for implementing Florete using embedding strategy is $1179649 = (2^{17} * 3^2 + 1)$. As Floret

is already faster than the Saber and embedded Saber, we firmly believe that applying the embedding technique to Floret will improve its speed even more.

None of the three methods described above can avoid applying the forward NTT transformation on the public matrix or polynomial. This is only possible in schemes which have been designed by keeping NTT-friendliness in mind where the random public matrix or polynomial can be assumed to be in NTT domain already. However, unlike the first two methods the last method can be used to skip some application of forward NTT to secret polynomial by storing the secret in the NTT domain without introducing large overhead for memory.

C.2 Sable

Applying the embedding strategy to Sable is very straightforward. All the techniques described by Chung *et al.* [CHK+20] can be applied to Sable without major changes. Moreover, due to small q and smaller secret distribution the embedding prime p is smaller in Sable. This offers smaller memory footprint and better efficiency than Saber. Similar to Florete we used the analysis described by Chung *et al.*'s analysis to calculate the embedding prime for Sable as shown in Table 7. This prime has been chosen such that it supports the incomplete-NTT as described in the original work, i.e., 6 layers of radix-2 NTT followed by a 4×4 schoolbook multiplication. We used the implementation provided by Chung *et al.* to implement Sable on a Cortex-M4 microcontroller. The results are presented in Table 8.

Scheme name		Sable			Saber		
Security level		Low	Medium	High	Low	Medium	High
Embedding prime	p	1049089	1574401	2100097	20972417	25166081	25166081
Representing bits	$\lceil log_2(prime) \rceil$	21	21	22	25	25	25

Table 7: Comparison of prime used in embedded Sable and embedded Saber.

Scheme Name	Cortex-M4 performance (X1000 clock cycles)			Cortex-M4 memory (bytes)		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Sable	547	741	723	26864	28504	29536
Saber	658	864	835	27440	29080	30176

Table 8: Performance comparison in Cortex-M4 implementation of embedded Sable with embedded Saber.

We also want to note that, here the implementation of Sable has been made with few changes in the Chung *et al.*'s [CHK+20] implementation. Therefore, we do not consider this implementation as fully optimized and it is possible to improve this code for both more efficiency and smaller memory foot print. Sable uses a smaller modulus q and a smaller centered binomial distribution parameter η than Saber that implies Sable requires a fewer amount of pseudo-random numbers than Saber. Moreover, the value of η is 1 in Sable, so we do not need `load_littleendian` function (used in implementation of Saber [DKRV19]) to sample the secret by following centered binomial distribution. These two facts have a major contribution to the speed improvement of Sable. Nevertheless, we presented the results here to demonstrate the benefits of our design.

C.3 Espada

For Espada, we can also apply the embedding technique. The embedding prime p for the parameter set of Espada as presented in Table 2 is $75497729 = (2^8 * 41 * 7193 + 1)$.

Again following the argument of Chung *et al.* [CHK+20] we can use this prime for 4 layers of radix-2 NTT followed by a 4×4 schoolbook multiplication. As we have shown in this work, Espada has the smallest memory footprint among all lattice-based KEMs. Applying embedding technique will further improve the memory footprint since we do not need to store the intermediate polyonomials and results as we need for 2-levels Karatsuba implementation used here. Further, we believe that the embedding technique can improve the speed also of Espada.

C.4 NTT friendly instantiations

For NTT friendly instantiations where the NTT-friendly prime moduli p is fixed during design phase, we think it will be interesting to an extension of Espada, i.e., a KEM based on Module-LWE problem where the length of the polynomials are 64. For the other two cases, there already exists schemes such as compact-LWE described by Alkim *et al.* [ABC19] and Kyber [ABD+21] which are very efficient and compact. One thing to note that, Kyber uses rounding to reduce the length of their ciphertext. This introduces some rounding errors. However, while calculating the security they do not consider this rounding this rounding error except for the lower security version of Kyber ($l = 2$) [ABD+21]. It will be interesting to see in future if these schemes can be improved by considering the rounding noise and applying the strategies described here into account while calculating the security.

D Sub-functions performances of our schemes in Cortex-M4

The Table 9 contains performance breakdown of all our schemes with Saber into two major sub-functions polynomial multiplication and hash evaluation, and the last column represents the clock cycles that is required to perform other operations. The only difference in `opt` version of all our schemes with the normal one is they use assembly routines to perform polynomial multiplication instead of simple C code. It is clearly visible that we got speed-up for Florete than Saber due to the fact that not only it requires less pseudo-randomness but also it needs less number of 256×256 polynomial multiplications in `KeyGen` and `Encaps`. This table also shows that the performance of Espada heavily affected because it requires almost 4 times more pseudo-random numbers and 2 times more 64×64 polynomial multiplication than Saber. It is comprehensible that we can receive a certain performance improvement for Espada by using a hash function which is faster than Keccack (ex. Chacha). Kindly note that the stack memory requirement of Espada is just 1/3 of the stack memory uses of Saber approximately. The last scheme of our suite Sable got speed improvement because it demands less amount of pseudo-random numbers than Saber.

E On the combination of Toom-Cook multiplications in Florete

Our Ring-LWR based KEM Florete requires 768×768 polynomial multiplication. One of our primary motives for designing this scheme was to reuse hardware and software modules developed for Saber's 256×256 polynomial multiplication. Hence, we used as Toom-Cook 3-way multiplication on top of Saber's Toom-Cook 4-way + Karatsuba + schoolbook multiplication algorithm. However, it is not the only way to perform 768×768 polynomial multiplication. We describe below 5 additional combinations of Toom-Cook, Karatsuba and schoolbook multiplication to perform this multiplication.

Scheme name	Cortex-M4 total performance (X1000 clock cycles)			Hashing evaluation in Cortex-M4 (X1000 clock cycles)			Polynomial multiplication in Cortex-M4 (X1000 clock cycles)			Others operations in Cortex-M4 (X1000 clock cycles)		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	1178	2292	3170	194	362	258	941	1882	2823	42	48	87
Florete (opt)	439	814	953	194	362	258	202	404	606	42	48	87
Espada	4938	5372	5513	1486	1654	1513	3289	3563	3838	161	153	161
Espada (opt)	2343	2568	2497	1486	1654	1513	645	699	753	210	213	230
Sable	2031	2706	3159	359	501	398	1618	2158	2697	53	46	63
Sable (opt)	745	1004	1028	359	501	398	348	464	580	37	38	49
Saber	846	1098	1112	460	615	499	348	464	580	37	18	31

Table 9: Sub-functions performances of our schemes in Cortex-M4. Basic polynomial multiplication uses assembly routine in the `opt` version of schemes.

1. Karatsuba then schoolbook (`NoToom` in Table 10)
2. Toom-Cook 3-way then Karatsuba then schoolbook (`Toom3` in Table 10)
3. Toom-Cook 4-way then Karatsuba then schoolbook (`Toom4` in Table 10)
4. Toom-Cook 4-way then Toom-Cook 3-way then Karatsuba then schoolbook (`Toom4Toom3` in Table 10)
5. Toom-Cook 3-way then Toom-Cook 4-way then Karatsuba then schoolbook (`Toom3Toom4` in Table 10)

Florete with different 768×768 polynomial multiplication	Cortex-M4 performance (X1000 clock cycles)			Cortex-M4 memory (bytes)			Cortex-M4 performance of one polynomial multiplication (clock cycles)
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	
NoToom	533	1003	1236	12464	12632	12632	296249
Toom3	463	862	1025	16536	16704	16704	226081
Toom4	487	911	1098	17828	17996	17996	250290
Toom4Toom3	443	824	967	18840	19008	19008	206936
Toom3Toom4	439	814	953	18252	18420	18420	202307

Table 10: Performance comparison in Cortex-M4 implementation of Florete with 5 different ways of 768×768 polynomial multiplication.

Kannwischer *et al.* [KRS18] provided a software package that can generate optimized assembly routines for different combinations of Toom-Cook, Karatsuba, and schoolbook multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4. This software can optimally select the number of levels of Karatsuba multiplication and the final schoolbook multiplication. This software has options to generate code to perform first 768×768 polynomial multiplication only. To generate the code for the fifth option, we used this software to generate the code to perform 256×256 multiplication using Toom-Cook 4-way, Karatsuba and schoolbook. We then used the Toom-Cook 3-way wrapper from option 2 to implement the 5-th option of 768×768 polynomial multiplication efficiently. The required clock cycles to perform these above mentioned 5 ways of 768×768 polynomial multiplication has been shown in the Table 10. This table also contains time and stack memory needed in the Cortex-M4 platform for Florete considering the above mentioned 5 different ways of 768×768 polynomial multiplication. Since, for Florete our goal was to optimize the speed while

keeping the memory usage as low as possible, the 5-th choice is more logical choice for 768×768 polynomial multiplication. However, we also note that the combination of Toom-Cook 4-way and Toom-Cook 3-way multiplication is very close in performance and memory usage compared to the other choice.

F Area utilization of the full HW/SW system

Scheme	Platform	f	LUTs	FFs	DSPs	BRAMs
Espada	Zedboard	125	8,567	7,869	48	9
Sable (high speed)	Zedboard	150	10,561	9,406	0	11
Florete/Sable (compact)	Zedboard	125	7,356	7,400	38	11

Table 11: Area requirements of the HW/SW implementations in the FPGA, including the processor and interface costs.

G Toom-Cook-3 and Toom-Cook-4 multiplication

Algorithm 4: The Toom-Cook 3-way algorithm

Data: Two polynomials $A^{768}(x)$ and $B^{768}(x)$ of length 768.
Result: $C^{1535}(x) = A^{768}(x) \times B^{768}(x)$, polynomial of length 1535.

1 //Rewrite the polynomial A^{768} and B^{768}, where $y = x^{256}$.
2 $A^{768}(x) = A(y) = a_0^{256} + a_1^{256} y + a_2^{256} y^2$
3 $B^{768}(x) = B(y) = b_0^{256} + b_1^{256} y + b_2^{256} y^2$
4 //Evaluation
5 $A_1^{256}(x) = A(0) = a_0$
6 $A_2^{256}(x) = A(1) = a_0 + a_1 + a_2$
7 $A_3^{256}(x) = A(-1) = a_0 - a_1 + a_2$
8 $A_4^{256}(x) = A(-2) = a_0 - 2 \cdot a_1 + 4 \cdot a_2$
9 $A_5^{256}(x) = A(\infty) = a_2$
10 $B_1^{256}(x) = B(0) = b_0$
11 $B_2^{256}(x) = B(1) = b_0 + b_1 + b_2$
12 $B_3^{256}(x) = B(-1) = b_0 - b_1 + b_2$
13 $B_4^{256}(x) = B(-2) = b_0 - 2 \cdot b_1 + 4 \cdot b_2$
14 $B_5^{256}(x) = B(\infty) = b_2$
15 //256×256 Multiplication
16 $c_0 = C_1^{256}(x) = C(0) =$ Toom-Cook-4$(A_1^{256}(x), B_1^{256}(x))$
17 $c_1 = C_2^{256}(x) = C(1) =$ Toom-Cook-4$(A_2^{256}(x), B_2^{256}(x))$
18 $c_2 = C_3^{256}(x) = C(-1) =$ Toom-Cook-4$(A_3^{256}(x), B_3^{256}(x))$
19 $c_3 = C_4^{256}(x) = C(-2) =$ Toom-Cook-4$(A_4^{256}(x), B_4^{256}(x))$
20 $c_4 = C_5^{256}(x) = C(\infty) =$ Toom-Cook-4$(A_5^{256}(x), B_5^{256}(x))$
21 //Interpolation
22 $c_1 = (c_1 - c_2)/2$
23 $c_2 = c_2 - c_0$
24 $c_3 = (c_3 - c_1)/3$
25 $c_3 = (c_2 - c_3)/2 + 2 \cdot c_4$
26 $c_2 = c_2 + c_1 - c_4$
27 $c_1 = c_1 - c_3$
28 Return $C^{1535}(x) = C(y) = c_0 + c_1 y + c_2 y^2 + c_3 y^3 + c_4 y^4$

Algorithm 5: Toom-Cook 4-way algorithm (Toom-Cook-4) [DKSRV18, BZ06]

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 $C(y) = w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7$;