SABER: Mod-LWR based KEM (Round 3 Submission)

Principal submitter

This submission is from the following team, listed in alphabetical order:

- Andrea Basso, University of Birmingham
- Jose Maria Bermudo Mera, KU Leuven, imec-COSIC
- Jan-Pieter D'Anvers, KU Leuven, imec-COSIC
- Angshuman Karmakar, KU Leuven, imec-COSIC
- Sujoy Sinha Roy, University of Birmingham
- Michiel Van Beirendonck, KU Leuven, imec-COSIC
- Frederik Vercauteren, KU Leuven, imec-COSIC

E-mail address: `saber@esat.kuleuven.be`

Website: `https://www.esat.kuleuven.be/cosic/pqcrypto/saber/`

Telephone: +32-16-37-6080

Postal address:
Prof. Dr. Ir. Frederik Vercauteren
COSIC - Electrical Engineering
Katholieke Universiteit Leuven
Kasteelpark Arenberg 10
B-3001 Heverlee
Belgium

Auxiliary submitters: There are no auxiliary submitters.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: ×. See also printed version of "Statement by Each Submitter".

Document based on `pqskeleton` version 20170923 by Daniel Bernstein.

Changes with respect to Round 1 submission

Very few changes were made between the round 1 version and the round 2 version of Saber. The only changes made are as follows:

- Transposing matrix $\boldsymbol{A}$: in `Saber.PKE.KeyGen` given in Algorithm 1, the matrix $\boldsymbol{A}$ is now transposed in line 5. On the other hand, in `Saber.PKE.Enc` given in Algorithm 2 the matrix $\boldsymbol{A}$ is used without transpose in line 5. In the first round submission, this was the exact opposite: we used $\boldsymbol{A}$ in `KeyGen`, whereas $\boldsymbol{A}^T$ was used in `Enc`. The advantage of the new approach is that it allows to speed-up encryption.
- The parameter T: to simplify the description of the algorithms we introduced a parameter T which equals $2t$ in the first round submission. This has no impact on the actual implementation.
- Simplification of the specification: the round 2 version of Saber has a much simpler specification than the round 1 version by working entirely in the interval $[0, q[$ and never resorting to the centered interval $[-q/2, q/2]$. This has no impact on the actual implementation.
- The constant polynomial h has been removed and replaced by two new constant polynomials h_1 and h_2. This is needed to provably reduce the security of Saber to Mod-LWR and it slightly changes the implementation.

Changes with respect to Round 2 submission

- There are no changes to the specification of the scheme, all parameters are the same as in round 2.
- We updated the security estimates of Saber to account for an error in the Round 2 submission document as reported on the mailing list. To provide confidence the security estimate is calculated using three independent scripts.
- We thoroughly refactored the Saber reference implementation, making it easier to work with and reducing the codebase.
- We included an appendix containing two alternate instantiations of Saber (these are not part of the main submission):
  - Uniform Saber or uSaber: the distribution of the secret key vector is now uniformly random. This allows for more efficient secret key vector generation and might have advantages from a side-channel security point of view.
  - Saber-90s: replaces the hashing and pseudorandom number generating functions with well established functions for which there is widespread hardware support.

- Efficient implementations of Saber on a wide range of platforms have been added in Section 5.
- We have added a description of a *masked* implementation of Saber decapsulation on ARM Cortex-M4 in Section 5.2.1.
- We added a discussion on the concrete side-channel security of Saber in 6.4.

Contents

1 Introduction

Lattice based cryptography is one of the most promising cryptographic families that is believed to offer resistance to quantum computers. We introduce Saber [18], a family of cryptographic primitives that rely on the hardness of the Module Learning With Rounding problem (Mod-LWR). We first describe Saber.PKE, an IND-CPA secure encryption scheme, and transform it into Saber.KEM, an IND-CCA secure key encapsulation mechanism, using a version of the Fujisaki-Okamoto transform. The design goals of Saber were simplicity, efficiency and flexibility resulting in the following choices: all integer moduli are powers of 2 avoiding modular reduction and rejection sampling entirely; the use of LWR halves the amount of randomness required compared to LWE based schemes and reduces bandwidth; the module structure provides flexibility by reusing one core component for multiple security levels.

2 General algorithm specification (part of 2.B.1)

2.1 Notation

We denote with $\mathbb{Z}_q$ the ring of integers modulo an integer q with representants in $[0, q)$ and for an integer z, we denote with $z \bmod q$ the reduction of z in $[0, q)$. R_q is the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$ with n a fixed power of 2 (we only need $n = 256$). For any ring R, $R^{l\times k}$ denotes the ring of $l \times k$-matrices over R. For $p \mid q$, the mod p operator is extended to (matrices over) R_q by applying it coefficient-wise. Single polynomials are written without markup, vectors are bold lower case and matrices are denoted with bold upper case. If χ is a probability distribution over a set S, then $x \leftarrow \chi$ denotes sampling $x \in S$ according to χ. If χ is defined on $\mathbb{Z}_q$, $\boldsymbol{X} \leftarrow \chi(R_q^{l\times k})$ denotes sampling the matrix $\boldsymbol{X} \in R_q^{l\times k}$, where all coefficients of the entries in $\boldsymbol{X}$ are sampled from χ. The randomness that is used to generate the distribution can be specified as follows: $\boldsymbol{X} \leftarrow \chi(R_q^{l\times k}; r)$, which means that the coefficients of the entries in matrix $\boldsymbol{X} \in R_q^{l\times k}$ are sampled deterministically from the distribution χ using seed r. $\mathcal{U}$ denotes the uniform distribution and β_μ is a centered binomial distribution with parameter μ (where μ is even and the samples are in the interval $[-\mu/2, \mu/2]$) with probability mass function $P[x|x \leftarrow \beta_\mu] = \frac{\mu!}{(\mu/2+x)!(\mu/2-x)!} 2^{-\mu}$.

The bitwise shift operations $\ll$ and $\gg$ have the usual meaning when applied to an integer and are extended to polynomials and matrices by applying them coefficient-wise.

2.2 Parameter space

The parameters for Saber are:

- n, l: The degree $n = 256$ of the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$ and the rank l of the module which determine the dimension of the underlying lattice problem as $l \cdot n$.

Increasing the dimension of the lattice problem increases the security, but reduces the correctness.

- q, p, T: The moduli involved in the scheme are chosen to be powers of 2, in particular $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$ and $T = 2^{\epsilon_T}$ with $\epsilon_q > \epsilon_p > \epsilon_T$, so we have $T \mid p \mid q$. A higher choice for parameters p and T, will result in lower security, but higher correctness. A python script that calculates optimal values for p and T is part of the submission.

- μ: The coefficients of the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ are sampled according to a centered binomial distribution $\beta_\mu(R_q^{l\times 1})$ with parameter μ, where $\mu < p$. A higher value for μ will result in a higher security, but a lower correctness of the scheme.

- $\mathcal{F}$, $\mathcal{G}$, $\mathcal{H}$: The hash functions that are used in the protocol. Functions $\mathcal{F}$ and $\mathcal{H}$ are implemented using SHA3-256 (with specified input length), while $\mathcal{G}$ is implemented using SHA3-512.

- gen: The extendable output function that is used in the protocol to generate a pseudorandom matrix $\boldsymbol{A} \in R_q^{l\times l}$ from a seed $seed_{\boldsymbol{A}}$. It is implemented using SHAKE-128. It might be possible that a non-cryptographic pseudorandomness generator or a SHAKE-128 variant with a limited number of rounds suffices for security, which would speed up computations. However, as a thorough security evaluation of these options lacks, the more conservative SHAKE-128 is chosen.

2.3 Constants

The algorithm uses three constants: a constant polynomial $h_1 \in R_q$ with all coefficients set equal to $2^{\epsilon_q-\epsilon_p-1}$, a constant vector $\boldsymbol{h} \in R_q^{l\times 1}$ where each polynomial is equal to h_1 and a constant polynomial $h_2 \in R_q$ with all coefficients set equal to $(2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1} + 2^{\epsilon_q-\epsilon_p-1})$. These constants are used to replace rounding operations by a simple bit shift. The values of h_1 and h_2 are further tweaked to allow a provable reduction to the underlying Mod-LWR problem as explained in [22].

2.4 Saber Public Key Encryption

Saber.PKE is the public key encryption scheme consisting of the triplet of algorithms (Saber.PKE.KeyGen, Saber.PKE.Enc, Saber.PKE.Dec) as described in Algorithms 1, 2 and 3 respectively. The more detailed technical specifications are given in Section 8.

2.4.1 Saber.PKE Key Generation

The Saber.PKE key generation is specified by the following algorithm.

Algorithm 1: `Saber.PKE.KeyGen()`

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

2.4.2 Saber.PKE Encryption

The Saber.PKE Encryption is specified by the following algorithm, with optional argument r.

Algorithm 2: `Saber.PKE.Enc`$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in R_2; r)$

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

2.4.3 Saber.PKE Decryption

The Saber.PKE Decryption is specified by the following algorithm.

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

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

2.5 Saber Key-Encapsulation Mechanism

Saber.KEM is the key-encapsulation mechanism consisting of the triplet of algorithms (Saber.KEM.KeyGen, Saber.KEM.Enc, Saber.KEM.Dec) as described in Algorithms 4, 5 and 6 respectively. The more detailed technical specifications are given in Section 8.

2.5.1 Saber.KEM Key Generation

The Saber key generation is specified by the following algorithm.

Algorithm 4: `Saber.KEM.KeyGen()`

$(seed_{\boldsymbol{A}}, \boldsymbol{b}, \boldsymbol{s}) =$ `Saber.PKE.KeyGen()`
$pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
$pkh = \mathcal{F}(pk)$
$z = \mathcal{U}(\{0,1\}^{256})$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (z, pkh, pk, \boldsymbol{s}))$

2.5.2 Saber.KEM Key Encapsulation

The Saber key encapsulation is specified by the following algorithm and makes use of Saber.PKE.Enc as specified in Algorithm 2.

Algorithm 5: `Saber.KEM.Encaps`$(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}))$

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

2.5.3 Saber.KEM Key Decapsulation

The Saber key decapsulation is specified by the following algorithm and makes use of Saber.PKE.Dec as specified in Algorithm 3.

Algorithm 6: `Saber.KEM.Decaps`$(sk := (z, pkh, pk, \boldsymbol{s}))$

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

3 List of parameter sets (part of 2.B.1)

In this section we list the parameter sets for Saber.PKE and Saber.KEM, together with their security estimate, failure probability and bandwidth.

The security of Saber is expressed in core-SVP, i.e. based on only one execution of the SVP-oracle. We use the following estimates for the complexity of the state-of-the-art SVP solver in high dimensions as $2^{0.292b}$, which can be lowered to $2^{0.265b}$ using Grover's search algorithm, where b is the BKZ block size required. We report the security as estimated by the "leaky-LWE-estimator" [16], by the "estimate all the {LWE/NTRU} schemes" effort [2] and by a program written by D. J. Bernstein [11]. All estimators returned the same security in core-SVP up to small rounding differences.

The failure probability is estimated by exhaustively calculating the distribution of the error term and subsequently determining the probability of a failure as a sum of all values that exceed the decryption threshold. An exact formulation can be found in theorem 6.2 and python scripts to determine the failure probability are given in the submission package.

3.1 Saber.PKE parameter sets

For Saber.PKE, we define the following parameters sets with corresponding security levels in Table 1. The secret key can be compressed by only storing the $\lceil log_2(\mu + 1) \rceil$ LSB for each coefficient in the entries of $\boldsymbol{s}$. The values for a compressed secret key can be found in brackets.

Table 1: Security and correctness of Saber.PKE.

Security Category	Failure Probability	Classical Core-SVP	Quantum Core-SVP	pk (B)	sk (B)	ct (B)
LightSaber-PKE: $l = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^3$, $\mu = 10$						
1	2^{-120}	2^{118}	2^{107}	672	832 (256)	736
Saber-PKE: $l = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$, $\mu = 8$						
3	2^{-136}	2^{189}	2^{172}	992	1248 (384)	1088
FireSaber-PKE: $l = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^6$, $\mu = 6$						
5	2^{-165}	2^{260}	2^{236}	1312	1664 (384)	1472

3.2 Saber.KEM parameter sets

For Saber.KEM, we define the following parameters sets with corresponding security levels in Table 2. The secret key can be compressed by only storing the $\lceil log_2(\mu + 1) \rceil$ LSB for each coefficient in the entries of $\boldsymbol{s}$. The values for a compressed secret key can be found in

brackets. Note that only the secret key size (sk) differs from the Saber.PKE table due to the inclusion of the public key hash and the random value z.

Table 2: Security and correctness of Saber.KEM.

Security Category	Failure Probability	Classical Core-SVP	Quantum Core-SVP	pk (B)	sk (B)	ct (B)
LightSaber-KEM: $l = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^3$, $\mu = 10$						
1	2^{-120}	2^{118}	2^{107}	672	1568 (992)	736
Saber-KEM: $l = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$, $\mu = 8$						
3	2^{-136}	2^{189}	2^{172}	992	2304 (1440)	1088
FireSaber-KEM: $l = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^6$, $\mu = 6$						
5	2^{-165}	2^{260}	2^{236}	1312	3040 (1760)	1472

Core-SVP vs. Gates to break AES

To qualify for the different security categories (I, III, V), the NIST call for proposals states that "Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128/192/256-bit key" [30], and in particular, it lists the following number of classical gates for an attack on AES-128/192/256: 2^{143}, 2^{207} and 2^{272}.

The differences with the (classical) core-SVP metric given in the above tables are therefore 2^{25}, 2^{18} and 2^{12}, and labelling LightSaber as Category I thus relies on the assumption that this factor 2^{25} can be accounted for. At this moment it is simply impossible to provide a definite and precise answer to this question, and more research is required to settle this matter.

However, to illustrate that this is not an unreasonable assumption we provide the following non-asymptotic reasoning, where we take LightSaber as an example:

- The 2^{118} core-SVP was derived from dimension $\beta = 404$.
- Using the dimensions for free estimate $f = 11.5 + 0.075\beta = 42$ from [3] we require a sieve of dimension $362 = 404 - 42$.
- From the raw data accompanying [4], we obtain an estimate of 2^{134} gates for one iteration of the nearest neighbour search in dimension 362. We underestimate the total cost by ignoring the fact that more than one iteration is necessary in the sieve.
- Combining this with the estimate that BKZ-β requires $2d$ calls to SVP-β, i.e. roughly 2^{10} calls, we obtain an overall gate count of 2^{144} gates (where we ignored the required number of iterations within the sieve).

More precise estimates for the missing terms are required to assess the exact security margin.

4 Design rationale (part of 2.B.1)

Our design combines several existing techniques resulting in a very simple implementation, that reduces both the amount of randomness and the bandwidth required.

- Learning with Rounding (LWR) [6]: schemes based on (variants of) LWE require sampling from noise distributions which needs randomness. Furthermore, the noise is included in public keys and ciphertexts resulting in higher bandwidth (which can be mitigated by the use of compression techniques akin to LWR). In LWR based schemes, the noise is deterministically obtained by scaling down from a modulus q to a modulus p. This naturally reduces the size of the public keys and ciphertexts, lowers the overall number of secret polynomials that need to be sampled, and in turn reduces the time spent in hashing operations to generate pseudorandom bits. This becomes especially beneficial for masked implementations, where the overhead of both masking the hashing operation and masking the generation of the noise distribution is significantly higher than that of masking other operations.

- Modules [27, 14]: the module versions of the problems allow to interpolate between the original pure LWE/LWR problems and their ring versions, lowering computational complexity and bandwidth in the process. We use modules to protect against attacks on the ring structure of Ring-LWE/LWR and to provide flexibility. By increasing the rank of the module, it is easy to move to higher security levels without any need to change the underlying arithmetic.

- Encryption: we use a simple LWR version of Regev's LWE encryption scheme [35], where the encryption part is compressed (using the parameter T) to save on bandwith.

- Choice of moduli: all integer moduli in the scheme are powers of 2. This has several advantages: there is no need for explicit modular reduction; sampling uniformly modulo a power of 2 is trivial and thus avoids rejection sampling or other complicated sampling routines, which is important for constant time implementations; we immediately have that the moduli $p \mid q$ in LWR, which implies that the scaling operation maps the uniform distribution modulo q to the uniform distribution modulo p.

  The main disadvantage of using such moduli is that they do not natively support the number theoretic transform (NTT) for polynomial multiplication. In Saber, the polynomials are of small size (256 coefficients) and hence 'asymptotically' slower generic polynomial multiplication algorithms do not cause a noticeable slowdown and can even lead to faster results. Additionally, designers can choose the right polynomial multiplication algorithm depending on the platform, implementation strategy, and optimization goals.

  Note that in Saber we never require a multiplication of two random elements; we only require the multiplication of a random element by a small element. Instead of implementing this using general purpose multiplication techniques, this can also be implemented using simple shifts and additions/subtractions. Such approach is not

possible for submissions that work mostly in the NTT domain, since the smallness of elements is lost during the NTT. Finally, we remark that using a compression technique requires one to move back to the polynomial representation (the 'time domain'), so if low bandwidth is a design goal, a scheme that works purely in the NTT-domain ('frequency domain') is not possible.

5 Implementations and performance analysis (2.B.2)

Saber has been designed taking into account both security and implementation aspects. It is designed to be easy to understand and implement on a wide variety of platforms. Unnecessary complexities that could lead to dangerous implementation mistakes have been avoided to the best of our knowledge. Saber is constant-time by design and only uses simple operations. Therefore even a basic implementation of Saber will be relatively efficient and secure.

5.1 Performance on Intel Haswell platform

On modern Intel platforms, we can utilize AVX2 SIMD instructions to speedup Saber. Performance benchmark results of our AVX2-optimized implementation of Saber on Intel Xeon E3-1220 v3 (3100 MHz) Hiphop from Supercop [12] are shown in Table 3. The software includes optimization techniques from [18, 9].

Table 3: Cycle counts for AVX2-optimized implementation of Saber on Intel Xeon E3-1220 v3 (3100 MHz) Hiphop from Supercop [12].

Scheme	**Keygen**	**Encapsulation**	**Decapsulation**
LightSaber	45,232	62,236	62,624
Saber	80,340	103,204	103,092
FireSaber	126,220	153,832	155,700

Around 50-70% of the overall computation time in Saber is spent on pseudorandom number generation using the Keccak-based extendable output function SHAKE-128. On vector processing platforms, such as processors with SIMD, faster pseudorandom number generation is possible by using a vectorized implementation of SHAKE-128 along with multiple seeds. In Saber, a pseudorandom string is generated by calling SHAKE-128 serially, starting from a single seed. This design decision makes Saber's implementation easier and simpler on all kinds of platforms.

An additional implementation [37], which is called 'SaberX4', batches four Saber KEM operations and processes them simultaneously utilizing four 64-bit slots of AVX256. It achieves a higher throughput (i.e., more KEM operations per second) compared to the original Saber software on platforms with AVX2 SIMD support. SaberX4 can be used on a server platform

where high throughput (e.g., thousands of KEMs per second) is desired. Note that SaberX4 is not a variant of Saber; it is just a batched implementation of Saber (i.e., generates the same known answer test responses) that targets higher utilization of SIMD platforms. Performance improvements using SaberX4 are shown in Table 4. In a similar way, 'SaberX8' can be realized to achieve even higher throughput by batching eight Saber operations on state-of-the-art Intel platforms with AVX512. We anticipate that, in the future the relative latency as well as throughput of a batched Saber software will only improve with improvements in computer architecture technologies.

Table 4: Performance comparisons between SaberX4 and Saber on Intel Xeon E3-1220 v3 (3100 MHz) Hiphop from Supercop [12].

Scheme	**Operation**	**Latency** (cycles)	**Throughput** (ops/sec at 3.1GHz)
Saber	Keygen	80,340	38,586
`computes 1 KEM`	Encapsulation	103,204	30,037
	Decapsulation	103,092	30,070
SaberX4	Keygen	205,248	60,414
`computes 4 KEMs`	Encapsulation	251,248	49,353
	Decapsulation	271,096	45,740

5.2 Performance on ARM Cortex-M4 microcontroller

As discussed in Sec. 4, due to use of power-of-two moduli we can not use NTT based polynomial multiplication and therefore used a combination of Toom-Cook-Karatsuba-schoolbook polynomial multiplication. Since these techniques are asymptotically slower than NTT based polynomial multiplications, we devised techniques to speed-up this polynomial multiplication. Since the schoolbook multiplication is at the bottom of our hybrid multiplication scheme, it is called most frequently. In [25], we proposed a fast SIMD schoolbook multiplication using the DSP instructions of Cortex-M4 microcontrollers. Using this technique the number of instructions required to perform a single schoolbook multiplication decreases by 34%. We also proposed a vertical Toom-Cook evaluation and interpolation technique to speed-up our polynomial multiplication further at the expense of some extra memory usage. In [9], we also propose a lazy-interpolation based Toom-Cook polynomial multiplication and use precomputation to further speed-up our scheme.

Module-lattice based cryptography is known to require more storage to store their public-keys than their ring-lattice based counterparts. This is particularly problematic for small resource constrained devices such as Cortex-M4 microcontrollers. Most of this large storage requirement comes from the storage of public matrix $\boldsymbol{A}$ which in our case consists of l^2 polynomials of degree $n-1$. For example, for Saber ($l=3$) the memory required to store the public matrix is approximately 3.8kB. We proposed a *just-in-time* memory optimization technique that interleaves the Keccak squeeze stages to generate the polynomials one at a

time. This strategy reuses the memory required to hold the previous polynomial. Using this technique the space required for holding the public matrix $\boldsymbol{A}$ reduces to only 280 bytes. We also used an in-place Karatsuba multiplication to reduce the storage cost. We propose a compact representation of secret keys in Saber to reduce the size of secret keys and further reduce the storage cost.

Table 5 shows memory usage and cycles for the most balanced implementation of Saber in terms of speed and memory combining different optimization strategies described above.

Table 5: Cycle counts and RAM utilization of Saber on ARM Cortex-M4 microcontrollers

Scheme	**Operation**	**Latency** (cycles)	**Memory** (bytes)
LightSaber	Keygen	444,965	6,456
	Encapsulation	623,817	6,048
	Decapsulation	634,268	6,056
Saber	Keygen	846,136	7,488
	Encapsulation	1,098,472	6,560
	Decapsulation	1,112,393	6,568
FireSaber	Keygen	1,360,577	8,512
	Encapsulation	1,674,409	7,072
	Decapsulation	1,703,896	7,080

5.2.1 Masked implementation

We also targeted the ARM-Cortex M4 for a masked implementation of Saber decapsulation [7]. This implementation was shown secure against Differential Power Analysis (DPA) for 100,000 collected power traces using the well-known Test Vector Leakage Assessment (TVLA). More information on the concrete security of Saber against side-channel attacks is given in Section 6.4.

The masked implementation starts from the most balanced implementation of Saber in terms of speed and memory, which was described above. By using arithmetic masking for the polynomial operations, the polynomial multiplication and addition routines can be completely reused. Coefficient-wise shifting of polynomials used in the rounding operations are most easily protected using Boolean masking. Therefore, targeting Saber, we proposed a novel operation that integrates masked logical shifting together with arithmetic to Boolean mask conversion in a single operation. For masked binomial sampling we reused a sampler from [39] and modified it to additionally benefit from Saber's power-of-two moduli.

Ultimately, we showed that Saber is very efficient to mask, with only a factor 2.54× CPU cycles overhead and 1.77× increased dynamic memory consumption over an unmasked implementation as shown in Table 6. This is mostly due to two key properties of Saber. The

use of power-of-two moduli allows Saber to use simple and efficient conversion algorithms between Boolean and arithmetic masking, and Saber's choice for LWR replaces very costly masked noise sampling with significantly more efficient masked logical shifting. For comparison, a similar work targeting NewHope KEM — LWE with prime moduli — has a CPU cycles overhead factor of 5.74× [31].

Table 6: Cycles counts and RAM utilization of masked Saber on ARM Cortex-M4 microcontrollers.

Scheme	**Operation**	**Latency**		**Memory**	
		cycles	overhead	bytes	overhead
Saber	Masked Decapsulation	2,833,348	2.54×	11,656	1.77×

5.3 Performance on hardware platforms

The modular nature of Saber and use of power-of-two modului make Saber flexible and very efficient on hardware platforms. Two HW/SW codesigns [10, 17] and two complete HW [41, 46] implementations of Saber have been reported in the literature. Their speed and area requirements are reported in Table 7.

The first fully-in-hardware implementation [41] is public-domain. It takes advantage of the power-of-two moduli and small secret-size to realize a highly parallel polynomial multiplier (based on the schoolbook algorithm) that computes a polynomial multiplication in only 256 cycles. The implementation is an instruction-set coprocessor architecture that offers programmability and flexibility to computes all KEM operations (key generation, encapsulation and decapsulation) in hardware. The simple architecture of the polynomial multiplier and the flexibility of the coprocessor greatly simplifies the development of other implementations, such as a lightweight one.

The second hardware implementation [46] uses a tightly-coupled architecture and computes polynomial multiplication via the Karatsuba algorithm. This allows it to achieve very low computation times, at the cost of higher area consumption and less flexibility. In particular, the polynomial multiplier reduces the number of computations by up to 90% using an 8-level Karatsuba algorithm. This drastically reduces the cycle count, but the clock frequency is also lower. The same design was also implemented on an ASIC, where the clock frequency is higher and can achieve 4× performance compared to the FPGA results. Note that this implementation only computes Saber.PKE.

From a hardware implementation perspective, the serial execution of SHAKE-128 during pseudorandom number generation in the Saber algorithm does not cause any performance issues. Instead, it simplifies hardware implementations and reduces area requirements. This is due to the high speed of Keccak on hardware platforms [1]. For example, the public-domain high-speed implementation of the Keccak core by the Keccak Team [43] computes 'state-permutations' at a gap of only 28 cycles, thus generating 1,344 bits of pseudo-random

string every 28 cycles during the extraction-phase. Furthermore, one instance of the Keccak core consumes a large area, thus indicating that implementing a vectorized Keccak core would make the implementation both area-expensive and more complex. Additionally, as the Keccak core is already very fast in hardware, the use of parallel cores would be of little help in improving the speed in hardware.

Table 7: Performance of CCA-secure Saber (module dimension 3) on hardware platforms. Entries denoted by a * are estimated based on the timings of PKE functions.

Ref.	Platform	Time in μs (KeyGen./Encaps./ Decaps.)	Frequency (MHz)	Area (LUT/FF/DSP/BRAM) (or mm^2 for ASIC)
[10]	Artix-7 (HW/SW)	3.2K/4.1K/3.8K	125	7.4K/7.3K/28/2
[17]	UltraScale+ (HW/SW)	-/60/65	322	12.5K/11.6K/256/4
[41]	UltraScale+	21.8/26.5/32.1	250	23.6K/9.8K/0/2
[46]*	UltraScale+	-/14.0/16.8	100	34.9k/9.9K/85/6
[46]*	ASIC	2.60/3.49/4.21	400	0.35 mm^2

5.4 Saber on RSA coprocessor

Secure smart cards, microcontrollers, and trusted platform modules or hardware security modules come with dedicated coprocessors for accelerating RSA. Such coprocessors contain long integer multipliers for computing modular multiplications in the RSA algorithm. [44] accelerates polynomial multiplications in Saber using long integer multipliers. The authors report that, for polynomials with 256 coefficients, one polynomial multiplication takes 97K and 85K clock cycles for powers of 2 moduli 8192 and 1024 respectively on a ESP32 platform. Whereas, on the same platform NTT-based polynomial multiplications takes 244K clock cycles when the modulus is a prime 7681. Their work shows that Saber can benefit from long integer multipliers present in secured devices with RSA coprocessors.

6 Expected strength (2.B.4) in general

6.1 Security

The IND-CPA security of Saber.PKE can be reduced from the decisional Mod-LWR problem as shown by the following theorem:

Theorem 6.1. *In the random oracle model, where* `gen` *is assumed to be a random oracle, for any adversary A, there exist three adversaries B_0, B_1 and B_2 with roughly the same running time as A, such that* $Adv^{ind\text{-}cpa}_{Saber.PKE}(A) \leqslant Adv^{prf}_{\mathtt{gen}()}(B_0) + Adv^{mod\text{-}lwr}_{l,l,\nu,q,p}(B_1) + Adv^{mod\text{-}lwr}_{l+1,l,\nu,q,q/\zeta}(B_2)$, *where* $\zeta = \min(\frac{q}{p}, \frac{p}{T})$.

The correctness of Saber.PKE can be calculated using the python scripts included in the submission, following theorem 6.2:

Theorem 6.2. *Let* $\boldsymbol{A}$ *be a matrix in* $R_q^{l\times l}$ *and* $\boldsymbol{s}, \boldsymbol{s}'$ *two vectors in* $R_q^{l\times 1}$ *sampled as in Saber.PKE. Define* $\boldsymbol{e}$ *and* $\boldsymbol{e}'$ *as the rounding errors introduced by scaling and rounding* $\boldsymbol{A}^T\boldsymbol{s}$ *and* $\boldsymbol{A}\boldsymbol{s}'$*, i.e.* $((\boldsymbol{A}^T\boldsymbol{s}+\boldsymbol{h}) \bmod q) \gg (\epsilon_q-\epsilon_p) = \frac{p}{q}\boldsymbol{A}^T\boldsymbol{s}+\boldsymbol{e}$ *and* $((\boldsymbol{A}\boldsymbol{s}'+\boldsymbol{h}) \bmod q) \gg (\epsilon_q-\epsilon_p) = \frac{p}{q}\boldsymbol{A}\boldsymbol{s}'+\boldsymbol{e}'$*. Let* $e_r \in R_q$ *be a polynomial with uniformly distributed coefficients with range* $[-p/2T, p/2T]$*. If we set*

$$\delta = Pr[||(\boldsymbol{s}'^T\boldsymbol{e} - \boldsymbol{e}'^T\boldsymbol{s} + e_r) \mod p||_\infty > p/4]$$

then after executing the Saber.PKE protocol, both communicating parties agree on a n*-bit key with probability* $1-\delta$*.*

For these calculations, the failure probabilities of the different coefficients of $(\boldsymbol{s}'^T\boldsymbol{e}-\boldsymbol{e}'^T\boldsymbol{s}+e_r)$ can be assumed independent, as discussed in [21].

This IND-CPA secure encryption scheme is the basis for the IND-CCA secure KEM Saber.KEM=(`Encaps, Decaps`), which is obtained by using an appropriate transformation. Recently, several post-quantum versions [23, 42, 38, 24] of the Fujisaki-Okamoto transform with corresponding security reductions have been developed. At this point, the FO$^{\not\perp}$ transformation in [23] with post-quantum reduction from Jiang et al. [24] gives the tightest reduction for schemes with non-perfect correctness. However, other transformation could be used to turn Saber.PKE into a CCA secure KEM.

6.1.1 Security in the Random Oracle Model

By modeling the hash functions $\mathcal{G}$ and $\mathcal{H}$ as random oracles, a lower bound on the CCA security can be proven. We use the security bound of Hofheinz et al. [23], which considers a KEM variant of the Fujisaki-Okamoto transform that can also handle a small failure probability δ of the encryption scheme. This failure probability should be cryptographically negligibly small for the security to hold. Using Theorem 3.2 and Theorem 3.4 from [23], we get the following theorems for the security and correctness of our KEM in the random oracle model:

Theorem 6.3. *In the random oracle model, where* $\mathcal{G}$ *and* $\mathcal{H}$ *are assumed to be random oracles, for a IND-CCA adversary* B*, making at most* $q_\mathcal{H}$ *and* $q_\mathcal{G}$ *queries to respectively the random oracle* $\mathcal{G}$ *and* $\mathcal{H}$*, and* q_D *queries to the decryption oracle, there exists an IND-CPA adversary* A *with approximately the same running time as adversary* B*, such that:*

$$Adv^{ind\text{-}cca}_{Saber.KEM}(B) \leqslant 3Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + q_\mathcal{G}\delta + \frac{2q_\mathcal{G}+q_\mathcal{H}+1}{2^{256}}.$$

6.1.2 Security in the Quantum Random Oracle Model

Jiang et al. [24] provide a security reduction against a quantum adversary in the quantum random oracle model from IND-CCA security to OW-CPA security. IND-CPA with a

sufficiently large message space M implies OW-CPA [23, 13], as is given by following lemma:

Theorem 6.4. *For an OW-CPA adversary B, there exists an IND-CPA adversary A with approximately the same running time as adversary B, such that:*

$$Adv^{ow\text{-}cpa}_{Saber.PKE}(B) \leqslant Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + 1/|M|$$

Therefore, we can reduce the IND-CCA security of Saber.KEM from the IND.CPA security of the underlying public key encryption:

Theorem 6.5. *In the quantum random oracle model, where $\mathcal{G}$ and $\mathcal{H}$ are assumed to be random oracles, for any IND-CCA quantum adversary B, making at most $q_{\mathcal{H}}$ and $q_{\mathcal{G}}$ queries to respectively the random quantum oracle $\mathcal{G}$ and $\mathcal{H}$, and q_D many (classical) queries to the decryption oracle, there exists an adversary A, with approximately the same running time as B, such that:*

$$Adv^{ind\text{-}cca}_{Saber.KEM}(B) \leqslant 2q_{\mathcal{H}}\frac{1}{\sqrt{2^{256}}} + 4q_{\mathcal{G}}\sqrt{\delta} + 2(q_{\mathcal{G}} + q_{\mathcal{H}})\sqrt{Adv^{ind\text{-}cpa}_{Saber.PKE}(A) + 1/|M|}$$

In all attack scenarios we assume that the depth of quantum computation is limited to 2^{64} quantum gates.

Using recent results [26], it is likely that the square root loss can be avoided, but more research is required to evaluate whether Saber satisfies the conditions required by [26].

6.2 Multi-target protection

As described in [14], hashing the public key into $\hat{K}$ has two beneficial effects: it makes sure that K depends on the input of both parties, and it offers multi-target protection. Hashing pk into $\hat{K}$ ensures that an attacker is not able to use precomputed 'weak' values of m on multiple targets when searching for decryption failures as will be addressed in next section.

6.3 Decryption failure attack

Instead of solving the Mod-LWR problem, an attacker can mount an attack that uses decryption failures. In this scenario, the adversary uses Grover's algorithm to precompute m that have a relatively high failure probability. Once messages m are found that trigger a decryption failure, they can be used to estimate the secret. This attack strategy is covered by the $4q_{\mathcal{G}}\sqrt{\delta}$ term of the quantum IND-CCA security reduction.

The best known attack was described in [20] and the follow up work [19]. In a single target setting this attack requires an impractical number of decryption queries far above 2^{64}. It is possible to reduce the number of decryption queries required by the attack by doing additional preprocessing or by mounting a multi-target attack where a number of victims T

are targeted of which only one will be broken, in which case the attacker can perform $2^{64} \cdot T$ queries to find the first failing ciphertext. Note that due to the multi-target protection an attacker can not reuse ciphertexts over multiple victims and needs to generate each ciphertext independently. Both attacks are more costly than breaking the Mod-LWR problem of the public keys.

If future cryptanalysis in this direction would threaten the security of Saber, one could increase the parameter T to reduce the failure probability and thus increase the security at the cost of additional bandwidth.

6.4 Side-channel attacks

Timing analysis Information about the timing of certain calculations can be used to extract information about the long-term secret key. Constant-time execution is a well-known countermeasure against such attacks. As mentioned before, the design choices of Saber allow simple constant-time implementations. The use of power-of-two moduli avoids variable-time operations, such as rejection sampling or modular reduction with a prime modulus. All implementations of Saber described in Section 5 are constant-time.

Differential analysis Differential power or electromagnetic radiation analysis allows an attacker to infer the intermediate data that a device is processing through statistical analysis. These attacks are particularly powerful, requiring only minimal signal-to-noise ratios and limited knowledge of the underlying device. Such attacks are possible on all operations that are influenced by the long-term secret key (e.g., [33, 45]), notably also including the applications of error correcting codes and CCA-transform [34]. In a scenario where such attacks are possible, i.e. the attacker has physical access to the device, masked implementations are a popular countermeasure. Since Saber does not use any error correcting code, it avoids any complications related to masking an error correcting block.

We have described a first-order masked implementation of Saber decapsulation in Section 5.2.1, based on our work in [7]. The implementation is shown secure against first-order DPA on an STM32 microcontroller featuring the popular ARM Cortex-M4. Again, Saber is shown very efficient to protect due to its choice for power-of-two moduli. Additionally, for a masked implementation, the choice for rounding (LWR) instead of error sampling (LWE) offers additional efficiency.

In [7], we also briefly discuss extending the implementation to higher-order masking, since the first-order design is still vulnerable to higher-order differential analysis. We expect a higher-order masked implementation of Saber to benefit from its design choices in the same way, and to significantly outperform LWE-based schemes with prime moduli as well.

Single-trace attacks Implementations secure against differential analysis can still be broken using other side-channel attack methods. In particular, single-trace attacks can be used

to break masked implementations. These attacks process a power or EM trace horizontally, and the horizontal information of a single trace still contains information on both of the shares. Such attacks have recently been shown to be feasible on implementations of NewHope [5], Frodo [15], Kyber [32], and Saber [40]. Polynomial or matrix multiplication with the secret key are typical targets for these attacks. A popular countermeasure is randomized shuffling. Randomness is used to shuffle the order of operations or introduce dummy operations, which can be applied on top of masked implementations. These techniques increase the noise level to harden an implementation against higher-order DPA attacks as well. Shuffling was included in the linear operations of a masked implementation of NewHope KEM [31] at an extra overhead cost of only $1.01\times$, but this does not yet protect against attacks that target the non-linear NTT [32]. For Saber, which uses a combination of Toom-Cook, Karatsuba and schoolbook multiplication, further research is necessary to assess the exact granularity and overall cost of including shuffling operations.

7 Advantages and limitations (2.B.6)

Advantages:

- No modular reduction: since all moduli are powers of 2 we do not require explicit modular reduction. Furthermore, sampling a uniform number modulo a power of 2 is trivial in that it does not require any rejection sampling or more complicated sampling routines. This is especially important when considering constant time implementations.
- Modular structure and flexibility: the core component consists of arithmetic in the fixed polynomial ring $\mathbb{Z}_{2^{13}}[X]/(X^{256}+1)$ for all security levels. To change security, one simply uses a module of higher rank.
- Less pseudorandomness required: due to the use of Mod-LWR, our algorithm requires less pseudorandomness since no error sampling is required as in (Mod-)LWE. More specifically, instead of generating $2k+1$ secret polynomials as would be the case in (Mod-)LWE schemes, our design only needs to generate k secret polynomials. This is especially beneficial in masked implementations, where due to the non-linearity the cost of the secret polynomial sampling increases sharply with the masking order.
- Use of powers of 2 moduli p and q, and T simplifies scaling and rounding operations significantly.
- Low bandwidth: again due to the use of Mod-LWR, the bandwidth required is lower than similar systems based on (Mod-)LWE.
- Generic polynomial multiplication: since Saber does not make any specific algorithm an integral part of the protocol, implementers can choose the best polynomial multiplication algorithm depending on the platform and application constraints. This has allowed efficient implementations of Saber on a wide range of platforms: high-end

platforms such as Intel Haswell, resource-constrained and low power microcontrollers such as Arm Cortex M4 and M0, smart cards containing RSA coprocessors, FPGA platforms, and ASIC.

- No full multiplications: all multiplications that occur in the algorithms are multiplying a random element in R_q by a small element sampled from $\beta_\mu(R_q)$. Since the small element has coefficients bounded by $\mu/2$ in absolute value, it is possible to replace the full multiplication of random elements in R_q by simple circular shifts and additions. We note that this is not possible when using NTT since the smallness of elements from $\beta_\mu(R_q)$ is lost due to the NTT.

- Good for anonymous communication: execution is constant-time over different public keys and communication is uniformly random. As a result of the power-of-two moduli, the public key and ciphertext of Saber are distributed as a uniformly random bitstring as opposed to schemes with prime-moduli, which have a specific distribution of these terms due to modular reduction. The lack of modular reduction additionally implies that Saber is naturally constant time over different public keys in contrast to prime-moduli schemes where the generation time of $\boldsymbol{A}$ will depend on the public key. [1]

- Very efficient masking: Saber's choice for Mod-LWR and power-of-two moduli proves to be very efficient to mask. Power-of-two moduli allow Saber to use simple and efficient mask conversion routines between arithmetic and Boolean masking. LWR as the underlying primitive replaces very costly masked noise sampling with significantly more efficient masked logical shifting.

Limitations:

- The use of two-power moduli makes NTT-like polynomial multiplication not natively supported. Asymptotically slower polynomial multiplication algorithms such as Toom-Cook, Karatsuba, Schoolbook, or hybrids of them are used in Saber. Since the polynomials are of small size (256 coefficients only), the asymptotic superiority of NTT over the above mentioned generic algorithms does not play a big role role in performance. On some platforms, generic algorithms even outperform NTT-based polynomial multiplications. Additionally, as stated earlier, there is no need for multiplying two random elements in R_q, so a complete polynomial multiplier is not strictly required.

- The functionality is limited to an encryption scheme and a KEM. No signature scheme is provided.

8 Technical Specifications (2.B.1)

This section provides technical specifications for implementing Saber. For more details, the reader may read the C source code present in the reference implementation package.

[1] We would like to thank Peter Schwabe for bringing this to our attention

8.1 Implementation constants

The values of the implementation constants used in the algorithms are provided in Table 8.

Table 8: Implementation constants

Constants	LightSaber	Saber	FireSaber
SABER_L	2	3	4
SABER_EQ	13	13	13
SABER_EP	10	10	10
SABER_ET	3	4	6
SABER_SEEDBYTES	32	32	32
SABER_NOISE_SEEDBYTES	32	32	32
SABER_KEYBYTES	32	32	32
SABER_HASHBYTES	32	32	32
SABER_INDCPA_PUBLICKEYBYTES	672	992	1312
SABER_INDCPA_SECRETKEYBYTES	832	1248	1664
SABER_PUBLICKEYBYTES	672	992	1312
SABER_SECRETKEYBYTES	1568	2304	3040
SABER_BYTES_CCA_DEC	736	1088	1472

8.2 Data Types and Conversions

8.2.1 Bit Strings and Byte Strings

A bit is an element of the set $\{0, 1\}$ and a bit string is an ordered sequence of bits. In a bit string the rightmost or the first bit is the least significant bit and the leftmost or the last bit is the most significant bit. A byte is a bit string of length 8 and a byte string is an ordered array of bytes. Following the same convention, the rightmost or the first byte is the least significant byte and the leftmost or the last byte is the most significant byte.

For example, consider the byte string of length three: $3d\ 2c\ 1b$. The most significant byte is $3d$ and the least significant byte is $1b$. This byte string corresponds to the bit string 0011 1101 0010 1100 0001 1011. The least significant bit of the byte string is 1 and the most significant bit is 0.

8.2.2 Concatenation of Bit Strings

Concatenation of two bit strings b_0 to b_1 is denoted by $b_1 \parallel b_0$ where b_0 is present in the least significant part and b_1 is present in the most significant part. The length of the concatenated bit string is the sum of the lengths of b_0 and b_1.

Similarly concatenation of n bit strings b_0 to b_{n-1} is denoted by $b_{n-1} \parallel b_{n-2} \parallel \ldots \parallel b_1 \parallel b_0$ where b_0 is present in the least significant part and b_{n-1} is present in the most significant

part. Naturally the length of the concatenated bit string is the sum of the lengths of b_0 to b_{n-1}.

8.2.3 Concatenation of Byte Strings

Concatenation of two byte strings B_0 to B_1 is denoted by $B_1 \parallel B_0$ where B_0 is present in the least significant part and B_1 is present in the most significant part. The length of the concatenated byte string is the sum of the lengths of B_0 and B_1.

Similarly concatenation of n byte strings B_0 to B_{n-1} is denoted by $B_{n-1} \parallel B_{n-2} \parallel \ldots \parallel B_1 \parallel B_0$ where B_0 is present in the least significant part and B_{n-1} is present in the most significant part. Naturally the length of the concatenated byte string is the sum of the lengths of B_0 to B_{n-1}.

8.2.4 Polynomials

For a modulus $N = 2^k$ we denote with $R = \mathbb{Z}_N[x]/(x^n + 1)$ the polynomial ring modulo $x^n + 1$ with coefficients in $\mathbb{Z}_N$. We will only require $n = 256$, so such polynomials will be represented as an array of 256 elements in $\mathbb{Z}_N$. For N we will the following values:

- $N = q = 2^{13}$, so each coefficient occupies 13 bits
- $N = p = 2^{10}$, so each coefficient occupies 10 bits
- $N = T = 2^{\epsilon_T}$, so each coefficient occupies ϵ_T bits, depending on which version of Saber is implemented
- $N = 2$, so each coefficient occupies 1 bits

The i-th coefficient of a polynomial object, say pol, is accessed by $pol[i]$. In the following example

$$pol = c_{255}x^{255} + \ldots + c_1 x + c_0 \tag{1}$$

the constant coefficient c_0 is accessed by $pol[0]$ and the highest-degree (i.e. x^{255}) coefficient c_{255} is accessed by $pol[255]$.

- $\mathsf{SHIFTLEFT}_N$: This function takes a polynomial in R_N and shifts each coefficient to the left over s positions. The algorithm is shown in Alg. 7
- $\mathsf{SHIFTRIGHT}_N$: This function takes a polynomial in R_N and shifts each coefficient to the right over s positions. The algorithm is shown in Alg. 8

8.2.5 Vectors

A vector in $R_N^{l\times 1}$ is an ordered collection of l polynomials from R_N. The i-th element of a vector object, say $\boldsymbol{v} \in R_N^{l\times 1}$, is accessed by $\boldsymbol{v}[i]$, where $(0 \leq i \leq l-1)$.

Algorithm 7: Algorithm $\mathsf{SHIFTLEFT}_N$

Input: *pin*: polynomial in R_N, shift s
Output: *pout*: polynomial in R_N.

```
for (i = 0, i < 256, i = i + 1) do
    pout[i] = (pin[i] ≪ s)
return pout
```

Algorithm 8: Algorithm $\mathsf{SHIFTRIGHT}_N$

Input: *pin*: polynomial in R_N, shift s
Output: *pout*: polynomial in R_N.

```
for (i = 0, i < 256, i = i + 1) do
    pout[i] = (pin[i] ≫ s)
return pout
```

8.2.6 Matrices

A matrix in $R_N^{l \times m}$ is a collection of $l \times m$ polynomials in row-major order. The polynomial present in the i-th row and j-th column a matrix object, say $\boldsymbol{M}$, is accessed by $\boldsymbol{M}[i, j]$. Here $(0 \leq i \leq l-1)$ and $(0 \leq j \leq m-1)$.

8.2.7 Data conversion algorithms

The data conversion algorithms allow to map byte strings to elements or vectors of elements of the ring $\mathbb{Z}_N$ for $N = 2^k$. The different N we use in the algorithm were specified in Subsection 8.2.4.

- $\mathsf{BS2POL}_N$: This function takes a byte string of length $k \times 256/8$ where $N = 2^k$ and transforms it into a polynomial in R_N. The algorithm is shown in Alg. 9.

Algorithm 9: Algorithm $\mathsf{BS2POL}_N$

Input: BS: byte string of length $k \times 256/8$ with $N = 2^k$
Output: pol_N: polynomial in R_N

```
Interpret BS as a bit string of length k × 256.
Split it into bit strings each of length k and obtain (bs_255 ‖ ... ‖ bs_0) = BS.
for (i = 0, i < 256, i = i + 1) do
    pol_N[i] ← bs_i
return pol
```

- $\mathsf{POL}_N\mathsf{2BS}$: This function takes a polynomial from R_N and transforms it into a byte string of length $k \times 256/8$ with $N = 2^k$. The algorithm is shown in Alg. 10.

Algorithm 10: Algorithm $\mathsf{POL}_N\mathsf{2BS}$

Input: pol_N: polynomial in R_N
Output: BS: byte string of length $k \times 256/8$ with $N = 2^k$

1 Interpret the coefficients of pol_q as bit strings, each of length k.
2 Concatenate the coefficients and obtain the bit string $bs = (pol_N[255] \parallel \ldots \parallel pol_N[0])$ of length $k \times 256$.
3 Interpret the bit string bs as the byte string BS of length $k \times 256/8$.
4 **return** BS

- $\mathsf{BS2POLVEC}_N$: This function takes a byte string of length $l \times k \times 256/8$ with $N = 2^k$ and transforms it into a vector in $R_N^{l\times 1}$. The algorithm is shown in Alg. 11.

Algorithm 11: Algorithm $\mathsf{BS2POLVEC}_N$

Input: BS: byte string of length $l \times k \times 256/8$
Output: $\boldsymbol{v}$: vector into $R_N^{l\times 1}$

1 Split BS into l byte strings of length $k \times 256/8$ and obtain $(BS_{l-1} \parallel \ldots \parallel BS_0) = BS$
2 **for** $(i = 0,\ i < l,\ i = i + 1)$ **do**
3 $\quad \boldsymbol{v}[i] = \mathsf{BS2POL}_N(BS_i)$
4 **return** $\boldsymbol{v}$

- $\mathsf{POLVEC}_N\mathsf{2BS}$: This function takes a vector from $R_N^{l\times 1}$ and transforms it into a byte string of length $l \times k \times 256/8$ with $N = 2^k$. The algorithm is shown in Alg. 12.

Algorithm 12: Algorithm $\mathsf{POLVEC}_N\mathsf{2BS}$

Input: $\boldsymbol{v}$: vector in $R_N^{l\times 1}$
Output: BS: byte string of length $l \times k \times 256/8$

1 Instantiate the byte strings BS_0 to BS_{l-1} each of length $k \times 256/8$.
2 **for** $(i = 0,\ i < l,\ i = i + 1)$ **do**
3 $\quad BS_i = \mathsf{POL}_N\mathsf{2BS}(\boldsymbol{v}[i])$
4 Concatenate these byte strings and get the byte string $BS = (BS_{l-1} \parallel \ldots \parallel BS_0)$.
5 **return** BS

8.3 Supporting Functions

8.3.1 SHAKE-128

SHAKE-128, standardized in FIPS-202, is used as the extendable-output function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and

generates the output byte string of length 'output_length' in the byte array *output_byte_string* as described below.

$$\mathsf{SHAKE\text{-}128}(\textit{output\_byte\_string}, \text{output\_length}, \textit{input\_byte\_string}, \text{input\_length}) \tag{2}$$

8.3.2 SHA3-256

SHA3-256, standardized in FIPS-202, is used as a hash function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and generates the output byte string of length 32 in the byte array *output_byte_string* as described below. It is important that any implementation of Saber enforces the input lengths of SHA3-256 as specified in Table 8 to avoid domain-separation attacks [8].

$$\mathsf{SHA3\text{-}256}(\textit{output\_byte\_string}, \textit{input\_byte\_string}, \text{input\_length}) \tag{3}$$

8.3.3 SHA3-512

SHA3-512, standardized in FIPS-202, is used as a hash function. It receives the input byte string from the byte array *input_byte_string* of length 'input_length' and generates the output byte string of length 64 in the byte array *output_byte_string* as described below.

$$\mathsf{SHA3\text{-}512}(\textit{output\_byte\_string}, \textit{input\_byte\_string}, \text{input\_length}) \tag{4}$$

8.3.4 Modulo

The modulo operation $y = x \bmod q$ performs a coefficient-wise modulo operation on the input x as defined in Subsection 2.1. As the divisor q is a power of two in our design, this operation can be implemented as a bitmasking operation.

8.3.5 HammingWeight

This function returns the Hamming weight of the input bit string. For example,

$$w = \mathsf{HammingWeight}(a) \tag{5}$$

returns the Hamming weight of the input bit string a to the integer w. Naturally, HammingWeight always returns non-negative integers.

8.3.6 Randombytes

This function outputs a random byte string of a specified length. The following example shows how to use randombytes to generate a random byte string *seed* of length SABER_SEEDBYTES.

$$\mathsf{randombytes}(\mathit{seed}, \text{SABER\_SEEDBYTES})$$

8.3.7 PolyMul

This function performs polynomial multiplications in R_p and R_q. For two polynomials a and b in R_p, their product $c \in R_p$ is computed using PolyMul as follows:

$$c = \mathsf{PolyMul}(a, b, p)\,.$$

Similarly, for two polynomials a' and b' in R_q, their product $c' \in R_q$ is computed using PolyMul as follows:

$$c' = \mathsf{PolyMul}(a', b', q)\,.$$

8.3.8 MatrixVectorMul

This function performs multiplication of a matrix, say $\boldsymbol{M} \in R_q^{l\times l}$, and a vector $\boldsymbol{v} \in R_q^{l\times 1}$ and returns the product vector $\boldsymbol{mv} = \boldsymbol{M} * \boldsymbol{v} \in R_q^{l\times 1}$. The algorithm is described in Alg. 13. The function is used in the following way.

$$\boldsymbol{mv} = \mathsf{MatrixVectorMul}(\boldsymbol{M}, \boldsymbol{v}, q)$$

Algorithm 13: Algorithm MatrixVectorMul

Input: $\boldsymbol{M}$: matrix in $R_q^{l\times l}$,
$\boldsymbol{v}$: vector in $R_q^{l\times 1}$,
q: coefficient modulus
Output: $\boldsymbol{mv}$: vector in $R_q^{l\times 1}$

```
Instantiate polynomial object c
for (i = 0, i < l, i = i + 1) do
    c = 0
    for (j = 0, j < l, j = j + 1) do
        c = c + PolyMul(M[i, j], v[j], q)
    mv[i] = c mod q
return mv
```

8.3.9 InnerProd

This function takes a vector $\boldsymbol{v}_a \in R_p^{l\times 1}$ and a vector $\boldsymbol{v}_b \in R_p^{l\times 1}$ and computes the inner product of $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$, which is a polynomial $c \in R_p$. The algorithm is described in Alg. 14. The function is used in the following way.

$$c = \mathsf{InnerProd}(\boldsymbol{v}_a, \boldsymbol{v}_b, p)$$

Algorithm 14: Algorithm InnerProd

Input: $\boldsymbol{v}_a$: vector in $R_p^{l\times 1}$,
$\boldsymbol{v}_b$: vector in $R_p^{l\times 1}$,
p: coefficient modulus
Output: c: polynomial in R_p

$c \leftarrow 0$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
  $c = c + \mathsf{PolyMul}(\boldsymbol{v}_a[i], \boldsymbol{v}_b[i], p)$
return $c \bmod p$

8.3.10 Verify

This function compares two byte strings of the same length and outputs a binary bit. The output bit is '0' if the byte strings are equal; otherwise it is '1'. The following example shows how to use Verify to compare the byte strings BS_0 and BS_1 of length 'input_length'.

$$c = \mathsf{Verify}(BS_0, BS_1, input\_length) \tag{6}$$

If $BS_0 = BS_1$ then $c = 0$; otherwise $c = 1$.

8.3.11 GenMatrix

This function generates a matrix in $R_q^{l\times l}$ from a random byte string (called seed) of length `SABER_SEEDBYTES`. The steps are described in the algorithm `GenMatrix` in Alg. 15. The use of `GenMatrix` to generate the matrix $\boldsymbol{A} \in R_q^{l\times l}$ from the seed $\text{seed}_{\boldsymbol{A}}$ is as follows.

$$\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$$

8.3.12 GenSecret

This function takes a random byte string (called seed) of length `SABER_NOISE_SEEDBYTES` as input and outputs a secret which is a vector in $R_q^{l\times 1}$ with coefficients sampled from a

Algorithm 15: Algorithm GenMatrix for generation of matrix $\boldsymbol{A} \in R_q^{l \times l}$

Input: $seed_{\boldsymbol{A}}$: random seed of length SABER_SEEDBYTES
Output: $\boldsymbol{A}$: matrix in $R_q^{l \times l}$

Instantiate byte string object buf of length $l^2 \times n \times \epsilon_q/8$
SHAKE-128($buf, l^2 \times n \times \epsilon_q/8, seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
Split buf into $l^2 \times n$ equal bit strings of bit length ϵ_q and obtain $(buf_{l^2 n-1} \parallel \ldots \parallel buf_0) = buf$
$k = 0$
for $(i_1 = 0,\ i_1 < l,\ i_1 = i_1 + 1)$ **do**
    for $(i_2 = 0,\ i_2 < l,\ i_2 = i_2 + 1)$ **do**
        for $(j = 0,\ j < n,\ j{=}j{+}1)$ **do**
            $\boldsymbol{A}[i_1, i_2][j] = buf_k$
            $k = k + 1$
return $\boldsymbol{A} \in R_q^{l \times l}$

centered binomial distribution β_μ. The steps are described in the algorithm GenSecret in Alg. 16 The use of GenSecret to generate a secret $\boldsymbol{s} \in R_q^{l \times 1}$ from a random seed $seed_{\boldsymbol{s}}$ is shown as follows.

$$\boldsymbol{s} = \mathsf{GenSecret}(seed_{\boldsymbol{s}})$$

Algorithm 16: Algorithm GenSecret for generation of secret $\boldsymbol{s} \in R_q^{l \times 1}$

Input: $seed_{\boldsymbol{s}}$: random seed of length SABER_NOISE_SEEDBYTES
Output: $\boldsymbol{s}$: vector in R_q^l

Instantiate a byte string object buf of length $l \times n \times \mu/8$
SHAKE-128(buf, $l \times n \times \mu/8$, $seed_{\boldsymbol{s}}$, SABER_NOISE_SEEDBYTES)
Split buf into $2 \times l \times n$ bit strings of length $\mu/2$ bits and obtain $(buf_{2ln-1} \parallel \ldots \parallel buf_0) = buf$
$k = 0$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
    for $(j = 0,\ j < n,\ j = j + 1)$ **do**
        $\boldsymbol{s}[i][j] = \mathtt{HammingWeight}(buf_k) - \mathtt{HammingWeight}(buf_{k+1}) \bmod q$
        $k = k + 2$
return $\boldsymbol{s} \in R_q^l$

8.4 IND-CPA encryption

The IND-CPA encryption consists of 3 components,

- Saber.PKE.KeyGen, returns public key and the secret key to be used in the encryption.
- Saber.PKE.Enc, returns the ciphertext obtained by encrypting the message.
- Saber.PKE.Dec, returns a message obtained by decrypting the ciphrtext.

8.4.1 Saber.PKE.KeyGen

This function generates public and secret key pair as byte strings of length SABER_INDCPA_PUBKEYBYTES and SABER_INDCPA_SECRETKEYBYTES respectively. The details of Saber.PKE.KeyGen are provided in Alg. 17.

Algorithm 17: Algorithm Saber.PKE.KeyGen for IND-CPA public and secret key pair generation

Output: $PublicKey_{cpa}$: byte string of public key,
$SecretKey_{cpa}$: byte string of secret key

1 randombytes($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
2 SHAKE-128($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES, $seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
3 randombytes($seed_{\boldsymbol{s}}$, SABER_NOISE_SEEDBYTES)
4 $\boldsymbol{A}$ = GenMatrix($seed_{\boldsymbol{A}}$)
5 $\boldsymbol{s}$ = GenSecret($seed_{\boldsymbol{s}}$)
6 $\boldsymbol{b}$ = MatrixVectorMul($\boldsymbol{A}^T, \boldsymbol{s}, q$) + $\boldsymbol{h}$ mod q // Here $\boldsymbol{A}^T$ is transpose of $\boldsymbol{A}$
7 **for** ($i = 0$, $i < l$, $i = i + 1$) **do**
8 $\quad \boldsymbol{b}_p[i]$ = SHIFTRIGHT($\boldsymbol{b}[i]$, EQ − EP)
9 $SecretKey_{cpa}$ = POLVEC$_q$2BS($\boldsymbol{s}$)
10 pk = POLVEC$_p$2BS($\boldsymbol{b}_p$)
11 $PublicKey_{cpa} = seed_{\boldsymbol{A}} \parallel pk$
12 **return** ($PublicKey_{cpa}$, $SecretKey_{cpa}$)

8.4.2 Saber.PKE.Enc

This function receives a 256-bit message m, a random seed $seed_{enc}$ of length SABER_SEEDBYTES and the public key $PublicKey_{cpa}$ as the inputs and computes the corresponding ciphertext $CipherText_{cpa}$. The steps are described in Alg. 18.

8.4.3 Saber.PKE.Dec

This function receives Saber.PKE.Enc generated $CipherText_{cpa}$ and Saber.PKE.KeyGen generated $SecretKey_{cpa}$ as inputs and computes the decrypted message m. The steps are shown in Alg. 19.

Algorithm 18: Algorithm Saber.PKE.Enc for INC-CPA encryption

Input: m: message bit string of length 256,
$seed_{\boldsymbol{s'}}$: random byte string of length SABER_SEEDBYTES,
$PublicKey_{cpa}$: public key generated using Saber.PKE.KeyGen

Output: $CipherText_{cpa}$: byte string of ciphertext

Extract pk and $seed_{\boldsymbol{A}}$ from $PublicKey_{cpa} = (seed_{\boldsymbol{A}} \parallel pk)$
$\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$
$\boldsymbol{s'} = \mathsf{GenSecret}(seed_{\boldsymbol{s'}})$
$\boldsymbol{b'} = \mathsf{MatrixVectorMul}(\boldsymbol{A}, \boldsymbol{s'}, q) + \boldsymbol{h} \bmod q$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
  $\boldsymbol{b'}[i] = \mathsf{SHIFTRIGHT}(\boldsymbol{b'}[i], \mathtt{EQ} - \mathtt{EP})$
$\boldsymbol{b} = \mathsf{BS2POLVEC}_p(pk)$
$v' = \mathsf{InnerProd}(\boldsymbol{b}, \boldsymbol{s'} \bmod p, p)$
$m_p = \mathsf{BS2POL}_2(m)$
$m_p = \mathsf{SHIFTLEFT}(m_p, \mathtt{EP} - 1)$
$c_m = \mathsf{SHIFTRIGHT}(v' - m_p + h_1 \bmod p, \mathtt{EP} - \mathtt{ET})$
$CipherText_{cpa} = (\mathsf{POL}_T\mathsf{2BS}(c_m) \parallel \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{b'}))$
return $CipherText_{cpa}$

Algorithm 19: Algorithm Saber.PKE.Dec for IND-CPA decryption

Input: $CipherText_{cpa}$: byte string of ciphertext generated using Saber.PKE.Enc,
$SecretKey_{cpa}$: byte string of secret key generated using Saber.PKE.KeyGen

Output: m: decrypted message bit string of length 256

$\boldsymbol{s} = \mathsf{BS2POLVEC}_q(SecretKey_{cpa})$
$(c_m \parallel ct) = CipherText$
$c_m = \mathsf{BS2POL}_T(c_m)$
$c_m = \mathsf{SHIFTLEFT}(c_m, \mathtt{EP} - \mathtt{ET})$
$\boldsymbol{b'} = \mathsf{BS2POLVEC}_p(ct)$
$v = \mathsf{InnerProd}(\boldsymbol{b'}, \boldsymbol{s} \bmod p, p)$
$m' = \mathsf{SHIFTRIGHT}(v - c_m + h_2 \bmod p, \mathtt{EP} - 1)$
$m = \mathsf{POL}_2\mathsf{2BS}(m')$
return (m)

8.5 IND-CCA KEM

The IND-CCA KEM consists of 3 algorithms.

- Saber.KEM.KeyGen, returns public key and the secret key to be used in the key encapsulation.
- Saber.KEM.Encaps, this function takes the public key and generates a session key and the ciphertext of the seed of the session key.

- Saber.KEM.Decaps, this function receives the ciphertext and the secret key and returns the session key corresponding to the ciphertext.

8.5.1 Saber.KEM.KeyGen

This function returns the public key and the secret key in two separate byte arrays of size SABER_PUBLICKEYBYTES and SABER_SECRETKEYBYTES respectively. The function is described in Alg. 20.

Algorithm 20: Algorithm Saber.KEM.KeyGen for generating public and private key pair.

Output: $PublicKey_{cca}$: public key for encapsulation,
$SecretKey_{cca}$: secret key for decapsulation

1 $(PublicKey_{cpa}, SecretKey_{cpa})$ = Saber.PKE.KeyGen()
2 SHA3-256($hash\_pk$, $PublicKey_{cpa}$, SABER_INDCPA_PUBKEYBYTES)
3 randombytes(z, SABER_KEYBYTES)
4 $SecretKey_{cca} = (z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa})$
5 $PublicKey_{cca} = PublicKey_{cpa}$
6 **return** $(PublicKey_{cca}, SecretKey_{cca})$

8.5.2 Saber.KEM.Encaps

This function generates a session key and the ciphertext corresponding the key. The algorithm is described in Alg 21.

Algorithm 21: Algorithm Saber.KEM.Encaps for generating session key and ciphertext.

Input: $PublicKey_{cca}$: public key generated by Saber.KEM.KeyGen
Output: $SessionKey_{cca}$: session key,
$CipherText_{cca}$: cipher text corresponding to the session key

1 randombytes(m, SABER_KEYBYTES)
2 SHA3-256(m, m, SABER_KEYBYTES)
3 SHA3-256($hash\_pk$, $PublicKey_{cca}$, SABER_INDCPA_PUBKEYBYTES)
4 $buf = (hash\_pk \parallel m)$
5 SHA3-512(rk, buf, 2×SABER_KEYBYTES)
6 Split rk in two equal chunks of length SABER_KEYBYTES and obtain $(r \parallel k) = rk$
7 $CipherText_{cca}$ = Saber.PKE.Enc($m, r, PublicKey_{cca}$)
8 SHA3-256(r', $CipherText_{cca}$, SABER_BYTES_CCA_DEC)
9 $rk' = (r' \parallel k)$
10 SHA3-256($SessionKey_{cca}$, rk', 2×SABER_KEYBYTES)
11 **return** $(SessionKey_{cca}, CipherText_{cca})$

8.5.3 Saber.KEM.Decaps

This function returns a secret key by decapsulating the received ciphertext. The algorithm is described in Alg 22.

Algorithm 22: Algorithm Saber.KEM.Decaps for recovering session key from ciphertext

Input: $CipherText_{cca}$: cipher text generated by Saber.KEM.Encaps,
$SecretKey_{cca}$: public key generated by Saber.KEM.KeyGen
Output: $SessionKey_{cca}$: session key

Extract $(z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa}) = SecretKey_{cca}$
$m = \mathsf{Saber.PKE.Dec}(CipherText_{cca}, SecretKey_{cpa})$
$buf \leftarrow hash\_pk \parallel m$
$\mathsf{SHA3\text{-}512}(rk, buf, 2\times\texttt{SABER\_KEYBYTES})$
Split rk in two equal chunks of length SABER_KEYBYTES and obtain $(r \parallel k)$
$CipherText'_{cca} = \mathsf{Saber.PKE.Enc}(m, r, PublicKey_{cpa})$
$c = \mathsf{Verify}(CipherText'_{cca}, CipherText_{cca}, \texttt{SABER\_BYTES\_CCA\_DEC})$
$\mathsf{SHA3\text{-}256}(r', CipherText_{cca}, \texttt{SABER\_BYTES\_CCA\_DEC})$
if $c = 0$ **then**
  $temp = (r' \parallel k)$
else
  $temp = (r' \parallel z)$
$\mathsf{SHA3\text{-}256}(SessionKey_{cca}, temp, 2\times\texttt{SABER\_KEYBYTES})$
return $SessionKey_{cca}$

A Alternate Instantiations

In this appendix we explore the parameter space of Saber and report on two alternate instantiations of the Saber design: the first explores the parameter space of Saber and highlights some interesting parameter sets that merit further attention, e.g. from an efficiency and side-channel security point of view, whereas the second provides a 90s-version of our scheme.

A.1 Exploring the parameter space

In Figure 1, we have plotted the results of an exhaustive exploration of the Saber parameter space, where we vary q, p and T, and sample the secret key either from the binomial distribution $\beta_\mu(R_q^{l\times 1})$ for varying μ (plotted in blue) or from the centered uniform distribution $\mathcal{U}_u(R_q^{l\times 1})$ for varying u (plotted in red), where $\mathcal{U}_u$ denotes the uniform distribution over the range $[-2^{u-1}, 2^{u-1} - 1]$.

It is obvious that the main parameter that determines security is the rank l of the module being used, corresponding to the three clusters of parameter sets that are clearly visible. Within each group, there is a trade-off between security, failure probability and ciphertext size. A larger secret key variance results in higher security, but also in a higher failure probability. The parameter T can be used to lower decryption failures at the expense of larger ciphertexts.

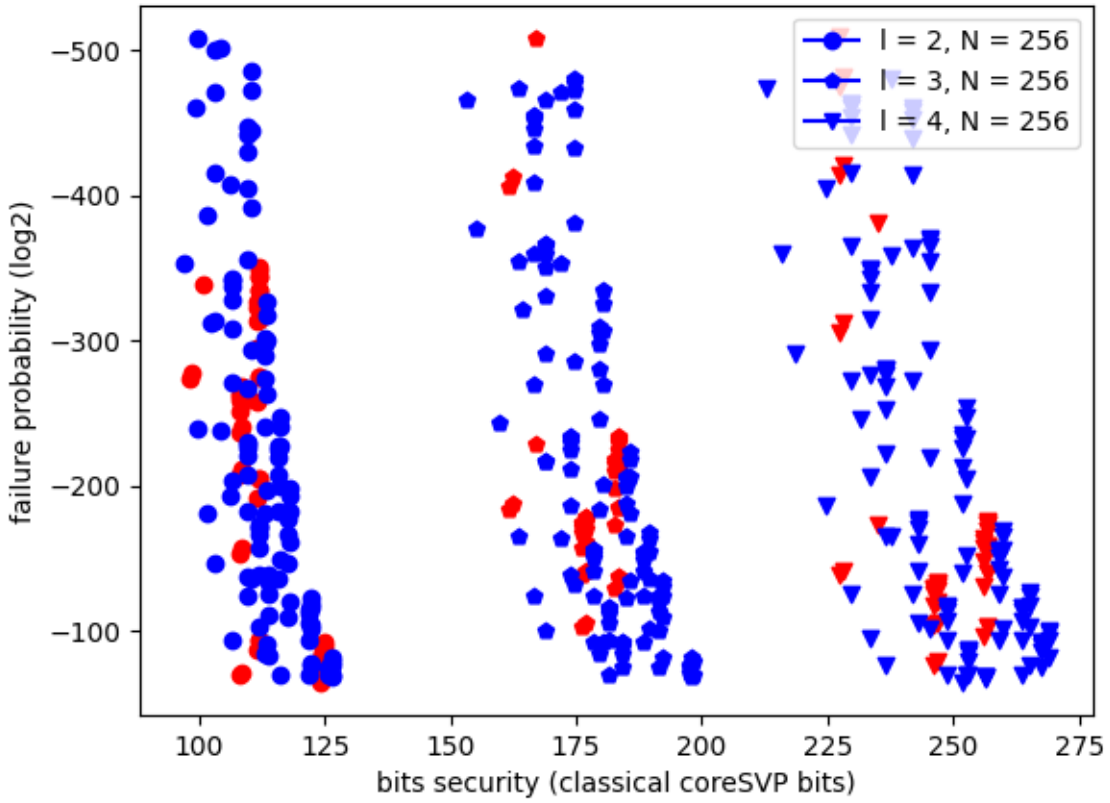

Figure 1: Parameter options for Saber. Binomial secret Saber parameter sets are in blue, uniform secret Saber parameter sets are in red.

A.2 uSaber

uSaber, or uniform-Saber, is a variant of Saber that samples the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ from the centered uniform distribution $\mathcal{U}_u(R_q^{l\times 1})$ instead of the binomial distribution $\beta_\mu(R_q^{l\times 1})$. Correspondingly, the secret coefficients in uSaber are in the interval $[-2^{u-1}, 2^{u-1}-1]$, instead of the interval $[-\mu/2, \mu/2]$.

The advantage of this choice is that the secret generation becomes more efficient as sampling from $\mathcal{U}_u$ is simpler than sampling from β_μ. Firstly, the number of pseudorandom bits required

for a similar secret variance is greatly reduced since a possible value for u is two bits, compared to six/eight/ten bits for μ. This has a beneficial impact on the time spent hashing. Secondly, the Hamming weight computation on the μ bits, previously given in Algorithm 16, is replaced by a simple sign extension of the u bits. Especially in a masked implementation where these two operations in masked form are increasingly dominant in the calculations, taking a uniform distribution results in a faster implementation. At the same time, the required masked primitive for uniform sampling is much simpler, resulting in an altogether more streamlined design. In hardware implementations, uniform sampling is faster and consumes a smaller area, mainly due to μ being a non-power-of-two in LightSaber and FireSaber.

The main problem of uSaber is that the availability of suitable parameter sets is limited. We propose to set $u = 2$ and thus sample the secrets coefficients from the interval $[-2, 1]$ for all three security levels, which are called uLightSaber, uSaber and uFireSaber. This has consequences on the other parameters, since a smaller range for the secret coefficients implies a lower security as well as a lower failure probability. To accommodate for this, we reduce the modulus q to 12-bits[2]. Note that uSaber offers slightly lower security than the regular variant of Saber, and it is currently unclear whether uLightSaber really satisfies Category I according to the classical gates requirement set out by NIST. Table 9 reports the parameters and the security of uLightSaber, uSaber and uFireSaber. The other parameters, l, n, p and T, are left unchanged. A reference implementation of uSaber is part of our submission in the "Variants" subdirectory. Table 10 contains a comparison of the performance of uSaber and regular Saber.

While uSaber is not part of the main submission at this point, it would be an interesting option if NIST decides that a classical coreSVP security of 2^{111} would be enough to fit security Category I. Also, in case of a mayor breakthrough in lattice reduction, a parameter set based on uSaber might be more efficient than a Saber based one.

Table 9: Security and correctness of uSaber KEM.

Security Category	Failure Probability	Classical Core SVP	Quantum Core SVP	pk (B)	sk (B)	ct (B)
uLightSaber-KEM: $l = 2$, $n = 256$, $q = 2^{12}$, $p = 2^{10}$, $T = 2^3$, $u = 2$						
1 (?)	2^{-184}	2^{111}	2^{101}	672	1504 (864)	736
uSaber-KEM: $l = 3$, $n = 256$, $q = 2^{12}$, $p = 2^{10}$, $T = 2^4$, $u = 2$						
3	2^{-167}	2^{182}	2^{165}	992	2208 (1248)	1088
uFireSaber-KEM: $l = 4$, $n = 256$, $q = 2^{12}$, $p = 2^{10}$, $T = 2^6$, $u = 2$						
5	2^{-152}	2^{256}	2^{232}	1312	2912 (1632)	1472

[2]further reducing the time spent on hashing operations, such as the pseudorandom generation of $\boldsymbol{A}$.

Table 10: Average cycle counts for reference implementations compiled with gcc 7.5.0 of Saber and uSaber on Intel Core i7-4510U CPU (2.00 GHz). Note that the comparison is between reference implementations, the comparison between optimized implementations may be partially different.

Scheme	**Keygen**	**Encapsulation**	**Decapsulation**
LightSaber	54,013	72,244	78,251
uLightSaber	48,234	63,800	69,979
Saber	107,339	132,889	149,678
uSaber	98,647	124,376	133,952
FireSaber	185,251	210,159	229,370
uFireSaber	155,112	188,131	201,991

A.3 Saber-90s

The software implementations of Saber spend the majority of their running time computing the Keccak function. By the time that post-quantum protocols are standardized and become mainstream, it is expected that there will be widespread hardware support for Keccak. Thus, the impact of Keccak and the overall running time of Saber will significantly decrease.

However, current implementations cannot yet utilize hardware-supported Keccak. It could be convenient, especially when experimenting with Saber in practical situations (e.g. within TLS), to replace the Keccak-based hashing and pseudorandom number generating functions with more well-established ones, such as SHA2 and AES. To avoid the spread of different and incompatible versions of Saber based on such functions, we propose a reference version. Following the trend set by other post-quantum schemes, such as Kyber-90s [36], we call this variant Saber-90s.

More in detail, we propose to use SHA2 to replace SHA3 (for both the 256 and 512 bits variants) and AES CTR_DRBG to replace SHAKE-128. This corresponds to replacing the following parameters:

- $\mathcal{F}, \mathcal{G}, \mathcal{H}$: The hash functions that are used in the protocol. Functions $\mathcal{F}$ and $\mathcal{H}$ are implemented using SHA2-256, while $\mathcal{G}$ is implemented using SHA2-512.
- gen: The extendable output function that is used in the protocol to generate a pseudorandom matrix $\boldsymbol{A} \in R_q^{l \times l}$ from a seed $seed_{\boldsymbol{A}}$. It is implemented using AES-256 in CTR mode, where $seed_{\boldsymbol{A}}$ is used as the key and the nonce is equal to 0. The counter of CTR mode is initialized to zero as well.

All the functions introduced in Saber-90s are NIST standards [29, 28]. Our submission package contains a reference implementation of Saber-90s in the "Variants" subdirectory.

Furthermore, since the goal of Saber-90s is to take advantage of dedicated hardware instructions we also provide an optimized implementation for modern Intel platforms with support for AVX2 instructions. Table 11 illustrates the performance advantage of such hardware support comparing the performance of an AVX2-optimized implementation of Saber and Saber-90s running on the same platform.

Table 11: Average cycle counts for AVX2-optimized implementations compiled with gcc 7.5.0 of Saber and Saber-90s on Intel Core i7-4510U CPU (2.00 GHz).

Scheme	**Keygen**	**Encapsulation**	**Decapsulation**
LightSaber	42,152	49,948	47,852
LightSaber-90s	28,928	35,491	35,123
Saber	66,727	79,064	76,612
Saber-90s	36,315	45,575	46,380
FireSaber	100,959	117,151	116,095
FireSaber-90s	57,144	70,335	72,797

References

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

[2] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate All the {LWE, NTRU} Schemes! In *Security and Cryptography for Networks - 11th International Conference, SCN 2018*, volume 11035 of *Lecture Notes in Computer Science*, pages 351–367. Springer, 2018.

[3] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *Advances in Cryptology - EUROCRYPT 2019 - Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019.

[4] Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. Estimating quantum speedups for lattice sieves. Cryptology ePrint Archive, Report 2019/1161, 2019. https://eprint.iacr.org/2019/1161.

[5] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating newhope with a single trace. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 189–205, Cham, 2020. Springer International Publishing.

[6] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom Functions and Lattices. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737. Springer, 2012.

[7] 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.

[8] Mihir Bellare, Hannah Davis, and Felix Günther. Separate your domains: Nist pqc kems, oracle cloning and read-only indifferentiability. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 3–32, Cham, 2020. Springer International Publishing.

[9] 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 Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, Mar. 2020.

[10] 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. *Accepted in DAC*, 2020:321, 2020.

[11] Daniel J. Bernstein. Personal communication, 2020.

[12] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/results-kem.html, "supercop-20200906" 2020.

[13] James Birkett and Alexander W. Dent. Relations Among Notions of Plaintext Awareness. In *Public Key Cryptography - PKC 2008*, pages 47–64, 2008.

[14] 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.

[15] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Assessing the feasibility of single trace power analysis of frodo. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 216–234, Cham, 2019. Springer International Publishing.

[16] 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*, pages 329–358, Cham, 2020. Springer International Publishing.

[17] 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. IEEE, 2019.

[18] 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 *AFRICACRYPT 2018*, pages 282–305, 2018.

[19] Jan-Pieter D'Anvers, Mélissa Rossi, and Fernando Virdia. (one) failure is not an option: Bootstrapping the search for failures in lattice-based encryption schemes. In *Advances in Cryptology – EUROCRYPT 2020*, pages 3–33. Springer, May 2020.

[20] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the impact of decryption failures on the security of LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1089, 2018. https://eprint.iacr.org/2018/1089.

[21] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on Ring/Mod-LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1172, 2018. https://eprint.iacr.org/2018/1172.

[22] 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. Cryptology ePrint Archive, Report 2018/230, 2018. https://eprint.iacr.org/2018/230.

[23] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A Modular Analysis of the Fujisaki-Okamoto Transformation. Cryptology ePrint Archive, Report 2017/604, 2017. http://eprint.iacr.org/2017/604.

[24] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. Post-quantum IND-CCA-secure KEM without Additional Hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[25] 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.

[26] Veronika Kuchta, Amin Sakzad, Damien Stehlé, Ron Steinfeld, and Shifeng Sun. Measure-rewind-measure: Tighter quantum random oracle model proofs for one-way to hiding and CCA security. In *Advances in Cryptology - EUROCRYPT 2020 - Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 703–728. Springer, 2020.

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

[28] National Institute of Standards and Technology. Advanced Encryption Standard (AES), 2001.

[29] National Institute of Standards and Technology. Secure Hash Standard (SHS), 2015.

[30] National Institute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[31] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):142–174, 2018.

[32] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 130–149, Cham, 2019. Springer International Publishing.

[33] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. Cryptology ePrint Archive, Report 2020/549, 2020. https://eprint.iacr.org/2020/549.

[34] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, Jun. 2020.

[35] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.

[36] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, Damien Stehlé. CRYSTALS-Kyber: Algorithm Specifications And Supporting Documentation (version 2.0), 2019.

[37] Sujoy Sinha Roy. SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism. In *37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019*, pages 321–324. IEEE, 2019.

[38] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model. Cryptology ePrint Archive, Report 2017/1005, 2017. https://eprint.iacr.org/2017/1005.

[39] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography – PKC 2019*, pages 534–564, Cham, 2019. Springer International Publishing.

[40] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Taeho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. Single-Trace Attacks on the Message Encoding of Lattice-Based KEMs. Cryptology ePrint Archive, Report 2020/992, 2020. https://eprint.iacr.org/2020/992.

[41] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, Aug. 2020.

[42] Ehsan Ebrahimi Targhi and Dominique Unruh. *Post-Quantum Security of the Fujisaki-Okamoto and OAEP Transforms*, pages 192–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[43] Keccak Team. Keccak in VHDL: High-speed Core. https://keccak.team/hardware.html, Accessed on November 2019.

[44] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 421–440, Cham, 2020. Springer International Publishing.

[45] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. Magnifying Side-Channel Leakage of Lattice-Based Cryptosystems with Chosen Ciphertexts: The Case Study of Kyber. Cryptology ePrint Archive, Report 2020/912, 2020. https://eprint.iacr.org/2020/912.

[46] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. A High-performance Hardware Implementation of Saber Based on Karatsuba Algorithm. Cryptology ePrint Archive, Report 2020/1037, 2020. https://eprint.iacr.org/2020/1037.