SABER: Mod-LWR based KEM (Round 2 Submission)

Principal submitter

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

- Jan-Pieter D'Anvers, KU Leuven, imec-COSIC
- Angshuman Karmakar, KU Leuven, imec-COSIC
- Sujoy Sinha Roy, 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. The principal submitter is the team listed above.

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: $\times$. See also printed version of "Statement by Each Submitter".

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

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 [5], 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 $p|q$, the $\bmod p$ 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:

- 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.

- n, l: The degree n and the number l of polynomials in the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s}'$ 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.

- $\mathcal{F}, \mathcal{G}, \mathcal{H}$: The hash functions that are used in the protocol. Functions $\mathcal{F}$ and $\mathcal{H}$ are implemented using SHA3-256, 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.

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 10.

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 $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}))$

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`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$

$\boldsymbol{A} = \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
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`$(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$

$v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
$m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
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 10.

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 $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}, z, pkh))$

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`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$

$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c =$ `Saber.PKE.Enc`$(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
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`$(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$

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

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

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 $log_2(\mu)$ LSB for each coefficient in the entries of $\boldsymbol{s}$. The values for a compressed secret key can be found in brackets.

Sec Cat	fail prob	attack	Classical	Quantum	pk (B)	sk (B)	ciphertext (B)
LightSaber-PKE: $l = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^3$, $\mu = 10$							
1	2^{-120}	primal dual	125 169	114 153	672	832(256)	736
Saber-PKE: $l = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$, $\mu = 8$							
3	2^{-136}	primal dual	203 244	185 226	992	1248(288)	1088
FireSaber-PKE: $l = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^6$, $\mu = 6$							
5	2^{-165}	primal dual	283 338	257 308	1312	1664(384)	1472

Table 1: Security and correctness of Saber.PKE.

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 $log_2(\mu)$ 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.

Sec Cat	fail prob	attack	Classical	Quantum	pk (B)	sk (B)	ciphertext (B)
LightSaber-KEM: $l = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^3$, $\mu = 10$							
1	2^{-120}	primal dual	125 169	114 153	672	1568(992)	736
Saber-KEM: $l = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$, $\mu = 8$							
3	2^{-136}	primal dual	203 244	185 226	992	2304(1344)	1088
FireSaber-KEM: $l = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^6$, $\mu = 6$							
5	2^{-165}	primal dual	283 338	257 308	1312	3040(1760)	1472

Table 2: Security and correctness of Saber.KEM.

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) [2]: 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 modulus p, which does not need randomness and naturally reduces bandwidth for keys and ciphertexts.

- Modules [12, 4]: 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. As in 'Kyber' [4], 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 [13], 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 it excludes the use of the number theoretic transform (NTT) to speed up polynomial multiplication. However, 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. Furthermore, we remark that 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 as in 'Kyber' 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 simply not possible.

5 Detailed performance analysis (2.B.2)

We evaluated the performance of the software implementation on a Dell laptop with an Intel(R) Core(TM) i7-6600U CPU 2.60GHz processor, Ubuntu operating system, and gcc compiler 7.0. We disabled hyperthreading and TurboBoost. The performance results for the various parameter sets of Saber.KEM can be found in Table 3. For performance on ARM microcontrollers please refer to [10].

Our key encapsulation mechanism uses three hash functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$. For hash functions $\mathcal{F}$ and $\mathcal{H}$, SHA3-256 is used, while $\mathcal{G}$ is implemented using SHA3-512.

Table 3: Performance of Saber.KEM. Cycles on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz for key generation, encapsulation, and decapsulation are represented by **K**, **E**, and **D** respectively in the 4th column. Sizes of secret key (sk), public key (pk) and ciphertext (c) are reported in the last column.

Scheme	Problem	Security	Cycles (ref)	Cycles (avx)	Bytes
LightSaber-KEM	Module-LWR	114	**K:** 85,474	61,849	**sk:** 1,568
			E: 108,927	72,692	**pk:** 672
			D: 119,868	70,605	**c:** 736
Saber-KEM	Module-LWR	185	**K:** 163,333	104,177	**sk:** 2,304
			E: 196,705	122,086	**pk:** 992
			D: 215,733	120,464	**c:** 1,088
FireSaber-KEM	Module-LWR	257	**K:** 259,504	161,379	**sk:** 3,040
			E: 308,277	184,766	**pk:** 1,312
			D: 341,654	186,625	**c:** 1,472

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. *For any adversary A, there exist three adversaries B_0, B_1 and B_2 such that $Adv^{ind\text{-}cpa}_{Saber.PKE}(A) \leqslant Adv^{prf}_{\mathbf{gen}()}(B_0) + Adv^{mod\text{-}lwr}_{l,l,\nu,q,p}(B_1) + Adv^{mod\text{-}lwr}_{l+1,l,\nu,q,q/\zeta}(B_2)$, 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 $\boldsymbol{As'}$*, 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* $((\boldsymbol{As'}+\boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) = \frac{p}{q}\boldsymbol{As'}+\boldsymbol{e'}$*. 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 [7].

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 [8, 15, 14, 9] of the Fujisaki-Okamoto transform with corresponding security reductions have been developed. At this point, the FO$^{\not\perp}$ transformation in [8] with post-quantum reduction from Jiang et al. [9] 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. [8], 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 [8], we get the following theorems for the security and correctness of our KEM in the random oracle model:

Theorem 6.3. *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 *such that:*

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

6.1.2 Security in the Quantum Random Oracle Model

Jiang et al. [9] 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 [8, 3], as is given by following lemma:

Theorem 6.4. *For an OW-CPA adversary* B*, there exists an IND-CPA adversary* A *such that:*

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

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. *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 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.

6.2 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 as described in [6]. This attack strategy is covered by the $4q_{\mathcal{G}}\sqrt{\delta}$ term of the quantum IND-CCA security reduction. However, the best known attack requires an impractical number of decryption queries far above 2^{64}.

6.3 Multi-target protection

As described in [4], 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.

7 Expected strength (2.B.4) for each parameter set

The expected strengths of Saber.PKE and Saber.KEM for each parameter set are included in Table 1 and Table 2.

8 Analysis of known attacks (2.B.5)

In this report, the security of Saber is based on only one execution of the SVP-oracle, which is a very conservative underestimation of the real security. Laarhoven [11] estimated the complexity for 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. We report the security as estimated by the "estimate all the {LWE/NTRU} schemes" effort [1] that bundles security estimates on all LWE/NTRU based submissions to round 1 of the NIST Post-Quantum Standardization Process.

9 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 randomness required: due to the use of Mod-LWR, our algorithm requires less randomness since no error sampling is required as in (Mod-)LWE.
- Low bandwidth: again due to the use of Mod-LWR, the bandwidth required is lower than similar systems based on (Mod-)LWE.
- 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.

Limitations:

- The use of two-power moduli precludes NTT-like multiplication algorithms, so we have to resort to Toom-Cook and Karatsuba. However, as the last advantage shows there is no need for multiplying two random elements in R_q, so a general polynomial multiplier is not strictly required.
- The functionality is limited to an encryption scheme and a KEM. No signature scheme is provided.

10 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.

10.1 Implementation constants

The values of the implementation constants used in the algorithms are provided in Table 4.

Table 4: Implementation constants

Constants	**LightSaber**	**Saber**	**FireSaber**
EQ	13	13	13
EP	10	10	10
ET	3	4	6
SABER_SEEDBYTES	32	32	32
SABER_INDCPA_PUBKEYBYTES	672	992	1312
SABER_INDCPA_SECRETKEYBYTES	832	1248	1664
SABER_NOISE_SEEDBYTES	32	32	32
SABER_PUBLICKEYBYTES	672	992	1312
SABER_SECRETKEYBYTES	1568	2304	3040
SABER_KEYBYTES	32	32	32
SABER_HASHBYTES	32	32	32
SABER_BYTES_CCA_DEC	736	1088	1472

10.2 Data Types and Conversions

10.2.1 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.

10.2.2 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}.

10.2.3 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}.

10.2.4 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

$$pol = c_{255}x^{255} + \ldots + c_1x + c_0 \tag{1}$$

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

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
```

- $\mathsf{SHIFTRIGHT}_N$: 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

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
```

10.2.5 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 $(0 \leq i \leq l-1)$.

10.2.6 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 $(0 \leq j \leq m-1)$.

10.2.7 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 10.2.4.

- $\mathsf{BS2POL}_N$: 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.
- $\mathsf{POL}_N\mathsf{2BS}$: 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 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 $k \times 256$.
Split it into bit strings each of length k and obtain $(bs_{255} \parallel \ldots \parallel bs_0) = BS$.
for *(*$i = 0$, $i < 256$, $i = i + 1$*)* **do**
    $pol_N[i] \leftarrow bs_i$
return pol

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 $N = 2^k$

Interpret the coefficients of pol_q as bit strings, each of length k.
Concatenate the coefficients and obtain the bit string $bs = (pol_N[255] \parallel \ldots \parallel pol_N[0])$ of length $k \times 256$.
Interpret the bit string bs as the byte string BS of length $k \times 256/8$.
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 $R_N^{l \times 1}$

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

10.3 Supporting Functions

10.3.1 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*

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

as described below.

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

10.3.2 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.

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

10.3.3 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.

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

10.3.4 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.

10.3.5 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.

10.3.6 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.

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

10.3.7 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:

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

10.3.8 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.

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

10.3.9 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.

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**
    $c = c + \mathsf{PolyMul}(\boldsymbol{M}[i, j], \boldsymbol{v}[j], q)$
  $\boldsymbol{mv}[i] = c \bmod q$
return $\boldsymbol{mv}$

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

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 $c \bmod p$

10.3.10 Verify

This function compares two byte strings of the same length and outputs a binary bit. The output bit is '1' if the byte strings are equal; otherwise it is '0'. 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 $c = 1$; otherwise $c = 0$.

10.3.11 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.

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

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$(buf, l^2 \times n \times \epsilon_q/8, seed_{\boldsymbol{A}}, \texttt{SABER_SEEDBYTES})$
Split buf into $l^2 \times n$ equal byte 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 $\boldsymbol{A} \in R_q^{l \times l}$

10.3.12 GenSecret

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

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

10.4 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.

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_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_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**
        $\boldsymbol{s}[i][j] = \texttt{HammingWeight}(buf_k) - \texttt{HammingWeight}(buf_{k+1}) \bmod q$
        $k = k + 2$
return $\boldsymbol{s} \in R_q^l$

10.4.1 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

randombytes($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
SHAKE-128($seed_{\boldsymbol{A}}$, SABER_SEEDBYTES, $seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
randombytes($seed_{\boldsymbol{s}}$, SABER_NOISE_SEEDBYTES)
$\boldsymbol{A} = \text{GenMatrix}(seed_{\boldsymbol{A}})$
$\boldsymbol{s} = \text{GenSecret}(seed_{\boldsymbol{s}})$
$\boldsymbol{b} = \text{MatrixVectorMul}(\boldsymbol{A}^T, \boldsymbol{s}, q) + \boldsymbol{h} \bmod q$ `// Here` $\boldsymbol{A}^T$ `is transpose of` $\boldsymbol{A}$
for $(i = 0,\ i < l,\ i = i + 1)$ **do**
    $\boldsymbol{b}_p[i] = \text{SHIFTRIGHT}(\boldsymbol{b}[i], \texttt{EQ} - \texttt{EP})$
$SecretKey_{cpa} = \text{POLVEC}_q\text{2BS}(\boldsymbol{s})$
$pk = \text{POLVEC}_p\text{2BS}(\boldsymbol{b}_p)$
$PublicKey_{cpa} = seed_{\boldsymbol{A}} \parallel pk$
return $(PublicKey_{cpa},\ SecretKey_{cpa})$

10.4.2 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.

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

1 Extract pk and $seed_{\boldsymbol{A}}$ from $PublicKey_{cpa} = (pk \parallel seed_{\boldsymbol{A}})$
2 $\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$
3 $\boldsymbol{s'} = \mathsf{GenSecret}(seed_{\boldsymbol{s'}})$
4 $\boldsymbol{b'} = \mathsf{MatrixVectorMul}(\boldsymbol{A}, \boldsymbol{s'}, q) + \boldsymbol{h} \bmod q$
5 **for** $(i = 0,\ i < l,\ i = i + 1)$ **do**
6 $\quad \boldsymbol{b'}[i] = \mathsf{SHIFTRIGHT}(\boldsymbol{b'}[i], \mathtt{EQ} - \mathtt{EP})$
7 $\boldsymbol{b} = \mathsf{BS2POLVEC}_p(pk)$
8 $v' = \mathsf{InnerProd}(\boldsymbol{b}, \boldsymbol{s'} \bmod p, p)$
9 $m_p = \mathsf{SHIFTLEFT}(m, \mathtt{EP} - 1)$
10 $c_m = \mathsf{SHIFTRIGHT}(v' - m_p + h_1 \bmod p, \mathtt{EP} - \mathtt{ET})$
11 $CipherText_{cpa} = (\mathsf{POL}_T\mathsf{2BS}(c_m) \parallel \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{b'}))$
12 **return** $CipherText_{cpa}$

10.4.3 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.

10.5 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.

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

$\boldsymbol{s} = \mathsf{BS2POLVEC}_q(SecretKey_{cpa})$

$(c_m \parallel ct) = CipherText$

$c_m = \mathsf{SHIFTLEFT}(c_m, \mathtt{EP} - \mathtt{ET})$

$\boldsymbol{b'} = \mathsf{BS2POLVEC}_p(ct)$

$v = \mathsf{InnerProd}(\boldsymbol{b'}, \boldsymbol{s} \bmod p, p)$

$m' = \mathsf{SHIFTRIGHT}(v - c_m + h_2 \bmod p, \mathtt{EP} - 1)$

$m = \mathsf{POL}_2\mathsf{2BS}(m')$

return (m)

10.5.1 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

$(PublicKey_{cpa}, SecretKey_{cpa}) = \mathsf{Saber.PKE.KeyGen}()$

$\mathsf{SHA3\text{-}256}(hash\_pk, PublicKey_{cpa}, \mathtt{SABER\_INDCPA\_PUBKEYBYTES})$

$\mathsf{randombytes}(z, \mathtt{SABER\_KEYBYTES})$

$SecretKey_{cca} = (z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa})$

$PublicKey_{cca} = PublicKey_{cpa}$

return $(PublicKey_{cca}, SecretKey_{cca})$

10.5.2 Saber.KEM.Encaps

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

10.5.3 Saber.KEM.Decaps

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

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

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

1 Extract $(z \parallel hash\_pk \parallel PublicKey_{cpa} \parallel SecretKey_{cpa}) = SecretKey_{cca}$
2 m = Saber.PKE.Dec($CipherText_{cca}$, $SecretKey_{cpa}$)
3 $buf \leftarrow hash\_pk \parallel m$
4 SHA3-512(kr, buf, 2×SABER_KEYBYTES)
5 Split kr in two equal chunks of length SABER_KEYBYTES and obtain $(r \parallel k)$
6 $CipherText'_{cca}$ = Saber.PKE.Enc($m, r, PublicKey_{cpa}$)
7 c = Verify($CipherText'_{cca}$, $CipherText_{cca}$, SABER_BYTES_CCA_DEC)
8 SHA3-256(r', $CipherText'_{cca}$, SABER_BYTES_CCA_DEC)
9 **if** $c = 0$ **then**
10 $temp = (k \parallel r')$
11 **else**
12 $temp = (z \parallel r')$
13 SHA3-256($SessionKey_{cca}$, $temp$, 2×SABER_KEYBYTES)
14 **return** $SessionKey_{cca}$

A 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.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Thijs Laarhoven. Search problems in cryptography. PhD thesis, Eindhoven University of Technology, 2015. http://www.thijs.com/docs/phd-final.pdf.

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

[13] 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.

[14] 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.

[15] 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.