SABER: Mod-LWR based KEM

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: `frederik.vercauteren@gmail.com`

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: ×. 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, 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. $\mathcal{U}$ denotes the uniform distribution and β_μ is a centered binomial distribution with parameter μ and corresponding standard deviation $\sigma = \sqrt{\mu/2}$. 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.

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. We use the part selection function $\texttt{bits}(x, i, j)$ with $x \in \mathbb{Z}_{2^k}$ and $k \leq j \leq i$ to access j consecutive bits of a positive integer x *ending* at the i-th index, producing an integer in $\mathbb{Z}_{2^j}$; i.e., written in C code the function returns $(x \gg (i - j))\&(2^j - 1)$. The part selection function is extended to polynomials and matrices by applying it coefficient-wise, where input polynomials are in R_{2^k} and output polynomials in R_{2^j}. Finally let $\lfloor\rceil$ denote rounding to the nearest integer, which is extended to polynomials and matrices 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 $\epsilon_q > \epsilon_p > (\epsilon_t + 1)$, so we have $2t \mid p \mid q$. 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, which is implemented using SHAKE-128.

2.3 Constants

The algorithm uses two constants: a constant polynomial $h \in R_q$ with all coefficients set equal to $(2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_t - 2})$ and a constant vector $\boldsymbol{h} \in R_q^{l\times 1}$ consisting of polynomials all coefficients of which are set to the constant $2^{\epsilon_q - \epsilon_p - 1}$. These constants are used to replace rounding operations by a simple bit select.

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()
1 $seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
3 $r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b} = \texttt{bits}(\boldsymbol{As} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
6 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($pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r$)
1 $\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
2 if $r$ is not specified then
3     $r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b'} = \texttt{bits}(\boldsymbol{A}^T \boldsymbol{s'} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
6 $v' = \boldsymbol{b}^T \texttt{bits}(\boldsymbol{s'}, \epsilon_p, \epsilon_p) \in R_p$
7 $c_m = \texttt{bits}(v' + 2^{\epsilon_p - 1} m, \epsilon_p, \epsilon_t + 1) \in R_{2t}$
8 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($sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'})$)
1 $v = \boldsymbol{b'}^T \texttt{bits}(\boldsymbol{s}, \epsilon_p, \epsilon_p) \in R_p$
2 $m' = \texttt{bits}(v - 2^{\epsilon_p - \epsilon_t - 1} c_m + h, \epsilon_p, 1) \in R_2$
3 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()
1 $seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
3 $r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b} = \texttt{bits}(\boldsymbol{As} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
6 $pkh = \mathcal{F}(seed_{\boldsymbol{A}}, \boldsymbol{b})$
7 $z = \mathcal{U}(\{0,1\}^{256})$
8 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($pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$)
1 $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
3 $c = \texttt{Saber.PKE.Enc}(pk, m; r)$
4 $K = \mathcal{H}(\hat{K}, c)$
5 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($sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c$)
1 $m' = \texttt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
2 $(\hat{K}', r') = \mathcal{G}(pkh, m')$
3 $c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
4 if $c = c'$ then
5     return $K = \mathcal{H}(\hat{K}', c)$
6 else
7     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: $k = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^2$, $\mu = 10$							
1	2^{-120}	primal	126	115	672	832(256)	736
		dual	126	115			
Saber-PKE: $k = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^3$, $\mu = 8$							
3	2^{-136}	primal	199	181	992	1248(288)	1088
		dual	198	180			
FireSaber-PKE: $k = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^5$, $\mu = 6$							
5	2^{-165}	primal	270	246	1312	1664(384)	1472
		dual	270	245			

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.

Sec Cat	fail prob	attack	Classical	Quantum	pk (B)	sk (B)	ciphertext (B)
LightSaber-KEM: $k = 2$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^2$, $\mu = 10$							
1	2^{-120}	primal	126	115	672	1568(992)	736
		dual	126	115			
Saber-KEM: $k = 3$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^3$, $\mu = 8$							
3	2^{-136}	primal	199	181	992	2304(1344)	1088
		dual	198	180			
FireSaber-KEM: $k = 4$, $n = 256$, $q = 2^{13}$, $p = 2^{10}$, $t = 2^5$, $\mu = 6$							
5	2^{-165}	primal	270	246	1312	3040(1760)	1472
		dual	270	245			

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) [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 modulus p, which does not need randomness and naturally reduces bandwidth for keys and ciphertexts.
- Modules [15, 8]: 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' [8], 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.
- Reconciliation: we use a simple reconciliation scheme [2] to reduce the failure rate significantly and therefore also the parameters.
- 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. We remark however 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

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 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	Bytes
LightSaber-KEM	Module-LWR	115	**K:** 105,881 **E:** 155,131 **D:** 179,415	**sk:** 1,568 **pk:** 672 **c:** 736
Saber-KEM	Module-LWR	180	**K:** 216,597 **E:** 267,841 **D:** 318,785	**sk:** 2,304 **pk:** 992 **c:** 1,088
FireSaber-KEM	Module-LWR	245	**K:** 360,539 **E:** 400,817 **D:** 472,366	**sk:** 3,040 **pk:** 1,312 **c:** 1,472

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

6.1 Security

The IND-CPA security of Saber.PKE can then be reduced to 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 $Adv^{ind\text{-}cpa}_{Saber.PKE}(A) \leqslant Adv^{prf}_{\texttt{gen()}}(B_0) + Adv^{mod\text{-}lwr}_{l,l,\nu,q,p}(B_1) + Adv^{mod\text{-}lwr}_{l+1,l,\nu,q,q/\zeta}(B_2)$, where $\zeta = \min\left(\frac{q}{p}, \frac{p}{2t}\right)$.*

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 $\boldsymbol{As}$ and $\boldsymbol{A}^T\boldsymbol{s}'$, i.e. $\texttt{bits}(\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}, \epsilon_q, \epsilon_p) = \frac{p}{q}\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{e}$ and $\texttt{bits}(\boldsymbol{A}^T\boldsymbol{s}' + \boldsymbol{h}, \epsilon_q, \epsilon_p) = \frac{p}{q}\boldsymbol{A}^T\boldsymbol{s}' + \boldsymbol{e}'$. Let $e_r \in R_q$ be a polynomial with uniformly distributed coefficients with range $[-p/4t, p/4t]$. 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 $1 - \delta$.

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 [11, 18, 16, 12] of the Fujisaki-Okamoto transform with corresponding security reductions have been developed. At this point, the $\text{FO}^{\not\perp}$ transformation in [11] with post-quantum reduction from Jiang et al. [12] 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. [11], 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 [11], 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:*

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

6.1.2 Security in the Quantum Random Oracle Model

Jiang et al. [12] 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 [11, 7], as is given by following lemma:

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

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

Therefore, we can reduce the IND-CCA security of Saber.KEM to 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:*

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

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

6.2 Multi target protection

As described in [8], 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. In this scenario, the adversary uses Grover's algorithm to precompute an m that has a relatively high failure probability. Hashing pk into $\hat{K}$ ensures that an attacker is not able to use precomputed 'weak' values of m.

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)

The vectors $\boldsymbol{b}, \boldsymbol{b}'$ and $\boldsymbol{c}$ are constructed using the secret $\boldsymbol{s}$ or $\boldsymbol{s'}$, and are publicly known. The security of the secrets needs to be guaranteed through the underlying Mod-LWR problem by choosing appropriate parameters, both for the main Mod-LWR construction with $\boldsymbol{b}$ and $\boldsymbol{b}'$, as for the reconciliation construction which generates $\boldsymbol{c}$.

Our security analysis is similar to the one in ‘a New Hope’ [3]. The hardness of Mod-LWR is analyzed as an LWE problem, since there are no known attacks that make use of the Module or LWR structure. A set of l LWR samples given by with $\boldsymbol{A} \leftarrow \mathcal{U}(R_q^{l\times l})$ and $\boldsymbol{s} \leftarrow \beta_\mu(R_q^{l\times 1})$, can be rewritten as an LWE problem in the following way:

$$\begin{aligned} &\left(\boldsymbol{A}, \left\lfloor \frac{p}{q}(\boldsymbol{As} \mod q) \right\rceil \mod p\right) \\ =&\left(\boldsymbol{A}, \frac{p}{q}(\boldsymbol{As} \mod q) + \boldsymbol{e} \mod p\right) \end{aligned}.$$

We can lift this to a problem modulo q by multiplying by $\frac{q}{p}$:

$$\frac{q}{p}\boldsymbol{b} = \boldsymbol{As} + \frac{q}{p}\boldsymbol{e} \mod q\,,$$

where $q/p\,\boldsymbol{e}$ is the random variable containing the error introduced by the rounding operation, of which the coefficients are discrete and nearly uniformly distributed in $(-q/2p, q/2p]$.

BKW type of attacks [13] and linearization attacks [4] are not feasible, since the number of samples is at most double the dimension of the lattice. Moreover, the secret vectors $\boldsymbol{s}$ and $\boldsymbol{s'}$ are dense enough to avoid the sparse secret attack described by Albrecht [1]. These attacks only remain infeasible if the generated secret vectors are timely refreshed. As a result, we end up with two main type of attacks: the primal and the dual attack, that make use of BKZ lattice reduction [9, 17].

BKZ uses an SVP oracle in lower dimension b to perform the lattice reduction. The running time of this oracle is exponential in the dimension of the lattice, and the oracle is executed a polynomial number of times. 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 [14] 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.

8.1 Weighted Primal Attack

The primal attack constructs a lattice that has a unique shortest vector that contains the noise $\boldsymbol{e}$ and the secret $\boldsymbol{s}$. BKZ, with a certain block dimension b, can be used to find this unique solution. An LWE sample $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ can be transformed to the following lattice: $\Lambda = \{\boldsymbol{v} \in \mathbb{Z}^{m+n+1} : (\boldsymbol{A}|\boldsymbol{I}_m|-\boldsymbol{b})\boldsymbol{v} = 0 \mod q\}$, with dimension $d = m+n+1$ and volume q^m. The unique shortest vector in this lattice is $\boldsymbol{v} = (\boldsymbol{s}, \boldsymbol{e}, 1)$, and it has a norm of $\lambda \approx \sqrt{n\sigma_s^2 + m\sigma_e^2}$. Using heuristic models, the primal attack succeeds if [3]:

$$\sqrt{n\sigma_s^2 + m\sigma_e^2} < \delta^{2b-d-1} \mathrm{Vol}(\Lambda)^{\frac{1}{d}}$$
$$\text{where: } \delta = ((\pi b)^{\frac{1}{d}} \frac{b}{2\pi e})^{\frac{1}{2(b-1)}}$$

However, the vector $\boldsymbol{v} = (\boldsymbol{s}, \boldsymbol{e}, 1)$ is unbalanced since $||\boldsymbol{s}_i||$ is not necessarily equal to $||\boldsymbol{e}_i||$. In our case, $||\boldsymbol{s}_i|| < ||\boldsymbol{e}_i||$, which can be exploited by the lattice rescaling method described by Bai et al. [5], and further analysed in [10]. The expected norm of each entry of $\boldsymbol{s}$ is σ_s, while the approximate expected norm of components of e_i is $\sigma_e = \sqrt{q^2/12p^2}$. We can therefore construct the weighted lattice:

$$\Lambda' = \{(\boldsymbol{x}, \boldsymbol{y}', z) \in \mathbb{Z}^n \times (\alpha^{-1}\mathbb{Z})^m \times \mathbb{Z} : (\boldsymbol{x}, \alpha\boldsymbol{y}', z) \in \Lambda\}$$
$$\text{where: } \alpha = \frac{\sigma_e}{\sigma_s}$$

with dimension $(n + m + 1)$ and volume $(q/\alpha)^m$. Analogous to [3], the primal attack is successful if the projected norm of the unique shortest vector on the last b Gram-Schmidt vectors is shorter than the $(d-b)^{\text{th}}$ Gram-Schmidt vector, or:

$$\sigma_s\sqrt{b} \leqslant \delta^{2b-d-1}\left(\frac{q}{\alpha}\right)^{\frac{m}{d}}.$$

8.2 Weighted Dual Attack

The dual attack tries to distinguish between an LWE sample $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ and a uniformly random sample by finding a short vector $(\boldsymbol{v}, \boldsymbol{w})$ in the lattice $\Lambda = \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{y} \mod q\}$. This short vector is used to compute a distinguisher $z = \boldsymbol{vb}$. If $\boldsymbol{b} = \boldsymbol{As} + \boldsymbol{e}$, we can write $z = \boldsymbol{vAs} + \boldsymbol{ve} = \boldsymbol{ws} + \boldsymbol{ve}$, which is small and approximately Gaussian distributed. If $\boldsymbol{b}$ is generated uniformly, z will also be uniform in q.

Since in our case, $||\boldsymbol{s}_i|| < ||\boldsymbol{e}_i||$, we observe that the $\boldsymbol{ws}$ term will be smaller than the $\boldsymbol{ve}$ term. The weighted attack optimizes the shortest vector so that these terms have a similar variance, by considering the weighted lattice $\Lambda' = \{(\boldsymbol{x}, \boldsymbol{y}') \in \mathbb{Z}^m \times (\alpha^{-1}\mathbb{Z})^n : (\boldsymbol{x}, \alpha\boldsymbol{y}') \in \Lambda \mod q\}$. This lattice has a dimension of $n+m$ and a volume of $(q/\alpha)^n$, so the BKZ algorithm will output a shortest vector $\boldsymbol{u} = (\boldsymbol{v}, \boldsymbol{w}') = (\boldsymbol{v}, \alpha\boldsymbol{w})$ of size $||\boldsymbol{u}|| \approx \delta^{m+n}(q/\alpha)^{\frac{n}{m+n}}$. Calculating z with the shortest vector in the LWE case gives $z = \boldsymbol{w}'(\alpha\boldsymbol{s}) + \boldsymbol{ve}$, of which the Gaussian distribution standard deviation will thus be equal to $\sigma_z = ||\boldsymbol{u}||\sigma_e = ||\boldsymbol{u}||\sqrt{q^2/12p^2}$.

Following the strategy of [3], we can now calculate the cost of the dual attack. The statistical distance between a uniformly distributed z and a Gaussian distributed z is bounded by $\epsilon = 4\exp(-2\pi^2\tau^2)$, where $\tau = \sigma_z/q$. Since the key is hashed, an advantage of ϵ is not sufficient and must be repeated at least $R = \max(1, 1/(2^{0.2075b}\epsilon^2))$ times. The cost of the dual attack is thus equal to:

$$\text{Cost}_{\text{dual}} = \text{Cost}_{\text{BKZ}} R = b2^{cb}R, .$$

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.

Limitations:

- The use of two-power moduli precludes NTT-like multiplication algorithms, so we have to resort to Toom-Cook and Karatsuba.
- 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 Data Types and Conversions

10.1.1 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.

10.1.2 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}.

10.1.3 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}.

10.1.4 Polynomials

All polynomials in $R_q = \mathbb{Z}_q[x]/(x^n+1)$ have 256 coefficients and the coefficients are of size 13 bits since $n = 256$ and $q = 2^{13}$. All polynomials in $R_p = \mathbb{Z}_p[x]/(x^n+1)$ have 256 coefficients and the coefficients are of size 10 bits since $n = 256$ and $p = 2^{10}$. 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 $pol[255]$.

10.1.5 Vectors

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

10.1.6 Matrices

A matrix in $R_q^{l\times m}$ 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 $(0 \leq j \leq m-1)$.

10.1.7 Data conversion algorithms

The data conversion algorithms are defined as follows.

- $\mathsf{BS2POL}_q$: This function takes a byte string of length $13 \times 256/8$ and transforms it into a polynomial in R_q. The algorithm is shown in Alg. 7.

Algorithm 7: Algorithm $\mathsf{BS2POL}_q$

```
  Input: $BS$: byte string of length $13 \times 256/8$
  Output: $pol_q$: polynomial in $R_q$
1 Interpret $BS$ as a bit string of length $13 \times 256$.
2 Split it into bit strings each of length 13 and obtain $(bs_{255} \parallel \ldots \parallel bs_0) = BS$.
3 for $(i = 0,\ i < 256,\ i = i+1)$ do
4     $pol_q[i] \leftarrow bs_i$
5 return $pol$
```

- $\mathsf{POL}_q\mathsf{2BS}$: This function takes a polynomial from R_q and transforms it into a byte string of length $13 \times 256/8$. The algorithm is shown in Alg. 8.
- $\mathsf{BS2POLVEC}_q$: This function takes a byte string of length $l \times 13 \times 256/8$ and transforms it into a vector in $R_q^{l\times 1}$. The algorithm is shown in Alg. 9.
- $\mathsf{POLVEC}_q\mathsf{2BS}$: This function takes a vector from $R_q^{l\times 1}$ and transforms it into a byte string of length $l \times 13 \times 256/8$. The algorithm is shown in Alg. 10.

```
Algorithm 8: Algorithm $\mathsf{POL}_q\mathsf{2BS}$
    Input: $pol_q$: polynomial in $R_q$
    Output: $BS$: byte string of length $13 \times 256/8$
1 Interpret the coefficients of $pol_q$ as bit strings, each of length 13.
2 Concatenate the coefficients and obtain the bit string $bs = (pol_q[255] \parallel \ldots \parallel pol_q[0])$ of
    length $13 \times 256$.
3 Interpret the bit string $bs$ as the byte string $BS$ of length $13 \times 256/8$.
4 return $BS$
```

```
Algorithm 9: Algorithm $\mathsf{BS2POLVEC}_q$
    Input: $BS$: byte string of length $l \times 13 \times 256/8$
    Output: $\boldsymbol{v}$: vector into $R_q^{l \times 1}$
1 Split $BS$ into $l$ byte strings of length $13 \times 256/8$ and obtain $(BS_{l-1} \parallel \ldots \parallel BS_0) = BS$
2 for $(i = 0,\ i < l,\ i = i + 1)$ do
3     $\boldsymbol{v}[i] = \mathsf{BS2POL}_q(BS_i)$
4 return $\boldsymbol{v}$
```

```
Algorithm 10: Algorithm $\mathsf{POLVEC}_q\mathsf{2BS}$
    Input: $\boldsymbol{v}$: vector in $R_q^{l \times 1}$
    Output: $BS$: byte string of length $l \times 13 \times 256/8$
1 Instantiate the byte strings $BS_0$ to $BS_{l-1}$ each of length $13 \times 256/8$.
2 for $(i = 0,\ i < l,\ i = i + 1)$ do
3     $BS_i = \mathsf{POLVEC}_q\mathsf{2BS}(\boldsymbol{v}[i])$
4 Concatenate these byte strings and get the byte string $BS = (BS_{l-1} \parallel \ldots \parallel BS_0)$.
5 return $BS$
```

- $\mathsf{BS2POL}_p$: This function takes a byte string of length $10 \times 256/8$ and transforms it into a polynomial in R_p. The algorithm is shown in Alg. 11.

```
Algorithm 11: Algorithm $\mathsf{BS2POL}_p$
    Input: $BS$: byte string of length $10 \times 256/8$
    Output: $pol_p$: polynomial in $R_p$
1 Interpret $BS$ as a bit string of length $10 \times 256$.
2 Split it into bit strings each of length 10 and obtain $(bs_{255} \parallel \ldots \parallel bs_0) = BS$.
3 for $(i = 0,\ i < 256,\ i = i + 1)$ do
4     $pol_p[i] \leftarrow bs_i$
5 return $pol$
```

- $\mathsf{POL}_p\mathsf{2BS}$: This function takes a polynomial from R_p and transforms it into a byte string of length $10 \times 256/8$. The algorithm is shown in Alg. 12.

```
Algorithm 12: Algorithm POL_pBS
  Input: $pol_p$: polynomial in $R_p$
  Output: $BS$: byte string of length $10 \times 256/8$
1 Interpret the coefficients of $pol_p$ as bit strings, each of length 10.
2 Concatenate the coefficients and obtain the bit string $bs = (pol_p[255] \parallel \ldots \parallel pol_p[0])$ of
   length $10 \times 256$.
3 Interpret the bit string $bs$ as the byte string $BS$ of length $10 \times 256/8$.
4 return $BS$
```

- $\mathsf{BS2POLVEC}_p$: This function takes a byte string of length $l \times 10 \times 256/8$ and transforms it into a vector $\boldsymbol{v} \in R_p^{l \times 1}$. The algorithm is shown in Alg. 13.

```
Algorithm 13: Algorithm BS2POLVEC_p
  Input: $BS$: byte string of length $l \times 10 \times 256/8$
  Output: $\boldsymbol{v}$: vector into $R_p^{l \times 1}$
1 Split $BS$ into byte strings of size $10 \times 256/8$ and obtain $(BS_{l-1} \parallel \ldots \parallel BS_0) = BS$
2 for ($i = 0,\ i < l,\ i = i + 1$) do
3     $\boldsymbol{v}[i] = \mathsf{BS2POL}_p(BS_i)$
4 return $\boldsymbol{v}$
```

- $\mathsf{POLVEC}_p\mathsf{2BS}$: This function takes a vector from $R_p^{l \times 1}$ and transforms it into a byte string of length $l \times 10 \times 256/8$. The algorithm is shown in Alg. 14.

```
Algorithm 14: Algorithm POLVEC_p2BS
  Input: $\boldsymbol{v}$: vector in $R_p^{l \times 1}$
  Output: $BS$: byte string of length $l \times 10 \times 256/8$
1 Instantiate the byte strings $BS_0$ to $BS_{l-1}$ each of length $10 \times 256/8$.
2 for ($i = 0,\ i < l,\ i = i + 1$) do
3     $BS_i = \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{v}[i])$
4 Concatenate these byte strings and get the byte string $BS = (BS_{l-1} \parallel \ldots \parallel BS_0)$.
5 return $BS$
```

- $\mathsf{MSG2POL}_p$: This function takes a 32 byte message and transforms it into a polynomial in R_p. The algorithm is shown in Alg. 15

Algorithm 15: Algorithm $\mathsf{MSG2POL}_p$

```
    Input: $m$: 32-byte message.
    Output: $pol_p$: polynomial in $R_p$.
1   Constant $const = \log_2(p) - 1$
2   Split $m$ into bit strings each of length 1 and obtain $(m_{255} \| \ldots \| m_0) = m$.
3   for ($i = 0,\ i < 256,\ i = i + 1$) do
4       $pol_p[i] = (m_i \ll const)$
5   return $pol_p$
```

10.2 Supporting Functions

10.2.1 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.

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

10.2.2 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.

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

10.2.3 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.

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

10.2.4 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.

10.2.5 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.

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

10.2.6 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.

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

10.2.7 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. 16. The function is used in the following way.

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

10.2.8 VectorMul

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 product of $\boldsymbol{v}_a^T$ and $\boldsymbol{v}_b$, which is a polynomial $c \in R_p$. Here $\boldsymbol{v}_a^T$ stands for the transpose of $\boldsymbol{v}_a$. The algorithm is described in Alg. 17. The function is used in the following way.

Algorithm 16: 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 $R_q^{l\times 1}$
1   Instantiate polynomial object $c$
2   for ($i = 0,\ i < l,\ i = i + 1$) do
3       $c = 0$
4       for ($j = 0,\ j < l,\ j = j + 1$) do
5           $c = c + \mathsf{PolyMul}(\boldsymbol{M}[i, j], \boldsymbol{v}[j], q)$
6       $\boldsymbol{mv}[i] = c$
7   return $\boldsymbol{mv}$
```

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

Algorithm 17: Algorithm VectorMul

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

10.2.9 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 $c = 0$; otherwise $c = 1$.

10.2.10 Round

This function takes a polynomial in R_q and rounds it into a polynomial in R_p. The steps are shown in Alg. 18. The following example shows the use of Round to transform a polynomial

$pol_q \in R_q$ into a polynomial $pol_p \in R_p$.

$$pol_p = \mathsf{Round}(pol_q)$$

Algorithm 18: Algorithm Round for rounding a polynomial $\in \mathbb{R}_q$

```
    Input: $pol_q$: polynomial $\in \mathbb{R}_q$
    Output: $pol_p$: polynomial $\in \mathbb{R}_p$
1 $const = \epsilon_q - \epsilon_p$
2 for ($i = 0,\ i < 256,\ i = i + 1$) do
3     $pol_p[i] = (pol_q[i] \gg const) \pmod{p}$
4 return $pol_p$
```

10.2.11 Floor

This function takes an input a and returns the largest integer less than or equal to a. The following example shows how the use of this function.

$$c = \mathsf{floor}(a)$$

For e.g., $c = 1$ when $a = \frac{3}{2}$ and $c = -2$ when $a = \frac{-3}{2}$.

10.2.12 ReconDataGen

This function takes a polynomial from R_p and generates the reconciliation vector. The steps performed in ReconDataGen are shown in Alg. 19. The following example shows how to use ReconDataGen to compute the reconciliation byte string rec from the input polynomial $pol_p \in R_p$.

$$rec = \mathsf{ReconDataGen}(pol_p)$$

10.2.13 Recon

This function takes a polynomial in R_p and associated reconciliation byte string as inputs and generates a bit string of length 256. The internal steps are shown in Alg. 20. The following example shows the use of Recon to generate the 256-bit bit string K from the reconciliation byte string rec and the polynomial $pol_p \in R_p$.

$$K = \mathsf{Recon}(rec, pol_p)$$

Algorithm 19: Algorithm ReconDataGen for generation of reconciliation data from a polynomial $\in \mathbb{R}_p$

```
    Input: $pol_p$: polynomial $\in \mathbb{R}_p$
    Output: $rec$: byte string of length (RECON_SIZE+1) $\times$ 256/8 bytes.
1   $const = \epsilon_p -$ RECON_SIZE $- 1$
2   Instantiate a vector object $\boldsymbol{v}$
3   for $(i = 0,\ i < 256,\ i = i + 1)$ do
4       $\boldsymbol{v}[i] = pol_p[i] \gg const$
5   Interpret the elements of $\boldsymbol{v}$ as bit strings, each of length RECON_SIZE+1 bits.
6   Concatenate the elements and obtain the bit string $bs = (\boldsymbol{v}[255] \| \ldots \| \boldsymbol{v}[0])$ of length
      (RECON_SIZE + 1) $\times$ 256.
7   Interpret the bit string $bs$ as the byte string $rec$ of length (RECON_SIZE + 1) $\times$ 256/8.
8   return $rec$
```

Algorithm 20: Algorithm Recon

```
    Input: $rec$: vector of 256 elements where each element is of RECON_SIZE+1 bits
           $pol_p$: polynomial $\in \mathbb{R}_p$
    Output: $K$: bit string of length 256
1   $const_0 = \epsilon_p -$ RECON_SIZE $- 1$
2   $const_1 = 2^{\epsilon_p - 2} - 2^{\epsilon_p - 2 - \mathtt{RECON\_SIZE}}$
3   for $(i = 0,\ i < 256,\ i = i + 1)$ do
4       $temp = rec[i] \ll const_0$
5       $temp = pol_p[i] - temp + const_1$
6       $K_i = \mathsf{floor}(\frac{temp}{2^{\log_2(p)-1}})$ // outputs a bit
7   return $K = (K_{255} \| \ldots \| K_0)$
```

10.2.14 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. 21. The use of GenMatrix to generate the matrix $\boldsymbol{A} \in R_q^{l \times l}$ from the seed seed$_{\boldsymbol{A}}$ is as follows.

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

10.2.15 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. 22 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.

Algorithm 21: 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 $R_q^{l \times l}$
 1 Instantiate byte string object $buf$ of length $l^2 \times n \times \epsilon_q/8$
 2 SHAKE-128($buf, l^2 \times n \times \epsilon_q/8, seed_{\boldsymbol{A}}$, SABER_SEEDBYTES)
 3 Split $buf$ into $l^2 \times n$ equal byte strings of bit length $\epsilon_q$ and obtain
     $(buf_{l^2n-1} \parallel \ldots \parallel buf_0) = buf$
 4 $k = 0$
 5 for ($i_1 = 0,\ i_1 < l,\ i_1 = i_1 + 1$) do
 6     for ($i_2 = 0,\ i_2 < l,\ i_2 = i_2 + 1$) do
 7         for ($j = 0,\ j < n,\ j{=}j{+}1$) do
 8             $\boldsymbol{A}[i_1, i_2][j] = buf_k$
 9             $k = k + 1$
10 return $\boldsymbol{A} \in R_q^{l \times l}$
```

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

Algorithm 22: 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 $R_q^2$
 1 Instantiate a byte string object $buf$ of length $l \times n \times \mu/8$
 2 SHAKE-128($buf$, $l \times n \times \mu/8$, $seed_{\boldsymbol{s}}$, SABER_SEEDBYTES)
 3 Split $buf$ into $2 \times l \times n$ bit strings of length $\mu/2$ bits and obtain
     $(buf_{2ln-1} \parallel \ldots \parallel buf_0) = buf$
 4 $k = 0$
 5 for ($i = 0,\ i < l,\ i = i + 1$) do
 6     for ($j = 0,\ j < n,\ j = j + 1$) do
 7         $\boldsymbol{s}[i][j]$ = HammingWeight($buf_k$) $-$ HammingWeight($buf_{k+1}$)
 8         $k = k + 2$
 9 return $\boldsymbol{s} \in R_q^2$
```

10.3 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.

10.3.1 Saber.PKE.KeyGen

This function generates IND-CPA 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. 23.

Algorithm 23: 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  $\boldsymbol{v}$ = MatrixVectorMul($\boldsymbol{A}^T, \boldsymbol{s}$) // Here $\boldsymbol{A}^T$ is transpose of $\boldsymbol{A}$
7  for ($i = 0,\ i < l,\ i = i + 1$) do
8      $\boldsymbol{v}_p[i]$ = Round($\boldsymbol{v}[i]$)
9  $SecretKey_{cpa}$ = POLVEC$_q$2BS($\boldsymbol{s}$)
10 $pk$ = POLVEC$_p$2BS($\boldsymbol{v}_p$)
11 $PublicKey_{cpa} = seed_{\boldsymbol{A}} \| pk$
12 return ($PublicKey_{cpa}$, $SecretKey_{cpa}$)
```

10.3.2 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. 24.

10.3.3 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. 25.

10.4 IND-CCA KEM

The IND-CCA KEM consists of 3 algorithms.

Algorithm 24: 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 $PublicKey_{cpa} = (pk \,\|\, seed_{\boldsymbol{A}})$
2 $\boldsymbol{A} = \mathsf{GenMatrix}(seed_{\boldsymbol{A}})$
3 $\boldsymbol{s'} = \mathsf{GenSecret}(seed_{\boldsymbol{s'}})$
4 $\boldsymbol{v} = \mathsf{MatrixVectorMul}(\boldsymbol{A}, \boldsymbol{s'})$
5 Instantiate vector object $\boldsymbol{v}_p \in R_p^{l\times 1}$
6 for $(i = 0,\ i < l,\ i = i + 1)$ do
7     $\boldsymbol{v}_p[i] = \mathsf{Round}(\boldsymbol{v}[i])$
8 $ct = \mathsf{POLVEC}_p\mathsf{2BS}(\boldsymbol{v}_p)$
9 $\boldsymbol{v'} = \mathsf{BS2POLVEC}_p(pk)$
10 $pol_p = \mathsf{VectorMul}(\boldsymbol{v'}, \boldsymbol{s'}, p)$
11 $m_p = \mathsf{MSG2POL}(m)$
12 $m_p = m_p + pol_p \mod p$
13 $rec = \mathsf{ReconDataGen}(m_p)$
14 $CipherText_{cpa} = (rec \,\|\, ct)$
15 return $CipherText_{cpa}$
```

Algorithm 25: 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
1 Output: $m$: decrypted message bit string of length 256
2 $\boldsymbol{s} = \mathsf{BS2POLVEC}_q(SecretKey_{cpa})$
3 Extract $(rec \,\|\, ct) = CipherText$
4 $\boldsymbol{b} = \mathsf{BS2POLVEC}_p(ct)$
5 $v' = \mathsf{VectorMul}(\boldsymbol{b}, \boldsymbol{s}, p)$
6 $m' = \mathsf{Recon}(rec, v')$
7 $m = \mathsf{POL2MSG}(m')$
8 return $(m)$
```

- 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.

10.4.1 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. 26.

Algorithm 26: 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 $(PublicKey_{cca}, SecretKey_{cca})$
```

10.4.2 Saber.KEM.Encaps

This function generates a session key and the ciphertext corresponding the key. The algorithm is described in Alg 27.

Algorithm 27: 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($kr$, $buf$, $2\times$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($SessionKey_{cca}$, $kr'$, $2\times$SABER_KEYBYTES)
11 return $(SessionKey_{cca}, CipherText_{cca})$
```

10.4.3 Saber.KEM.Decaps

This function returns a secret key by decapsulating the received ciphertext. The algorithm is described in Alg 28.

Algorithm 28: 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 $(z \| hash\_pk \| PublicKey_{cpa} \| SecretKey_{cpa}) = SecretKey_{cca}$
2  $m =$ Saber.PKE.Dec($CipherText_{cca}$, $SecretKey_{cpa}$)
3  $buf \leftarrow hash\_pk \| m$
4  SHA3-512($kr$, $buf$, $2\times$SABER_KEYBYTES)
5  Split $kr$ in two equal chunks of length SABER_KEYBYTES and obtain $(r \| k)$
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     $temp = (r' \| k)$
11 else
12     $temp = (z \| k)$
13 SHA3-256($SessionKey_{cca}$, $temp$, $2\times$SABER_KEYBYTES)
14 return $SessionKey_{cca}$
```

10.5 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
SABER_SEEDBYTES	32	32	32
RECON_SIZE	2	3	5
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

References

[1] Martin R. Albrecht. *On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL*, pages 103–129. Springer International Publishing, Cham, 2017.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation, 2016. http://cryptojedi.org/papers/#newhopesimple.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. Document ID: 0462d84a3d34b12b75e8f5e4ca032869, http://cryptojedi.org/papers/#newhope.

[4] Sanjeev Arora and Rong Ge. *New Algorithms for Learning in Presence of Errors*, pages 403–415. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[5] Shi Bai and Steven D. Galbraith. *Lattice Decoding Attacks on Binary LWE*, pages 322–337. Springer International Publishing, Cham, 2014.

[6] Abhishek Banerjee, Chris Peikert, and Alon Rosen. *Pseudorandom Functions and Lattices*, pages 719–737. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[7] James Birkett and Alexander W. Dent. Relations among notions of plaintext awareness. In *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, pages 47–64, 2008.

[8] 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.

[9] Yuanmi Chen and Phong Q. Nguyen. *BKZ 2.0: Better Lattice Security Estimates*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[10] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. Lizard: Cut off the tail! practical post-quantum public-key encryption from lwe and lwr. Cryptology ePrint Archive, Report 2016/1126, 2016. http://eprint.iacr.org/2016/1126.

[11] 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.

[12] 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.

[13] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 43–62, 2015.

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

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

[16] 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.

[17] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1):181–199, Aug 1994.

[18] 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.