Lizard Public Key Encryption

Submission to NIST proposal

- Name of the proposed cryptosystem : Lizard Public Key Encryption

- Principal submitter
  - Name : Jung Hee Cheon
  - E-mail address : jhcheon@snu.ac.kr
  - Telephone : +82 10-9767-1912
  - Organization : Seoul National University, Republic of Korea
  - Postal address :
    Department of Mathematical Sciences (Bldg 27, Rm 404),
    Seoul National University, Seoul, Republic of Korea
- Auxiliary submitters
  - Seoul National University : Jung Hee Cheon, Sangjoon Park, Joohee Lee, Duhyeong Kim, Yongsoo Song , Seungwan Hong, Dongwoo Kim, Jinsu Kim, Seong-Min Hong
  - Ulsan national institute of science and technology : Aaram Yun, Jeongsu Kim
  - Korea Internet & Security Agency : Haeryong Park, Eunyoung Choi, Kimoon Kim, Jun-Sub Kim, Jieun Lee
- Inventors
  - Jung Hee Cheon, Duhyeong Kim, Joohee Lee, Yongsoo Song
- Developers
  - Aaram Yun, Jeongsu Kim, Sangjoon Park, Dongwoo Kim, Seungwan Hong, Jinsu Kim, Seong-Min Hong, Haeryong Park, Eunyoung Choi, Kimoon Kim, Jun-Sub Kim, Jieun Lee
- Owner of the cryptosystem
  - Jung Hee Cheon, Duhyeong Kim, Joohee Lee, Yongsoo Song

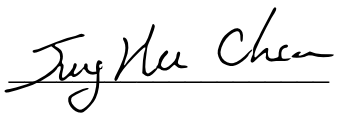
Principal submitter Jung Hee Cheon 

Table of Contents

1 Introduction

We propose Lizard, a family of post-quantum public-key encryption (PKE) schemes and key encapsulation mechanisms (KEMs).

At the center of our constructions lies the Lizard IND-CPA PKE. This is a scheme whose security is based on sparse, small secret versions of learning with errors (LWE) and learning with roundings (LWR). Essentially, the public-key is chosen to be a set of LWE samples with signed binary secrets, and the encryption uses rounding to achieve security (via LWR) and reduced size of the ciphertexts. We use sparse random vectors as ephemeral secrets to speed up multiplications. Our construction is based on a result we have proved that (sparse) signed binary secret LWE and LWR are at least as hard as the original LWE.

The IND-CPA PKE scheme is then turned into an IND-CCA2 KEM, via a KEM variant of the Fujisaki-Okamoto transformation. Using the usual KEM/DEM hybrid paradigm, this can be turned into an IND-CCA2 PKE scheme, for example by using the one-time pad to symmetrically encrypt messages with the symmetric key encapsulated by the KEM.

Also, we propose ring-based versions of the above constructions, called RLizard. Instead of based on the variants of LWE and LWR in Lizard, RLizard is based on the corresponding versions of ring-LWE and ring-LWR. As with Lizard, we first construct an IND-CPA PKE, and then obtain IND-CCA2 KEM and PKE by the same transformation.

1.1 Terminology and Notation

In this subsection, we introduce a list for terminology and notation used throughout this document.

log	the logarithm with base 2
n	the dimension of LWE samples, a positive integer
m	the number of LWE samples, a positive integer, a power of two
q	the large modulus, a positive integer, a power of two
p	the small modulus for rounding, a positive integer, a power of two
ℓ	a positive integer, the number of secret vectors in case of Lizard primitive, *i.e.* the number of plaintext slots in case of Lizard primitive
ℓ_1	a positive integer, the number of secret vectors in case of IND-CCA2 KEM schemes
ℓ_2	a positive integer, the number of ephemeral secret vectors in IND-CCA2 KEM schemes
d	a positive integer, the number of plaintext slots in case of IND-CCA2 PKE, the bit-length of shared secret key in case of IND-CCA2 KEM
h_s	the Hamming weight of a secret polynomial s
h_r	the Hamming weight of an ephemeral secret vector $\mathbf{r}$ or polynomial r
`_16_LOG_Q`	$16 - \log q$
`_16_LOG_T`	15
$\mathbb{Z}_q$	a set $\{0, 1, ..., q-1\}$
$\mathbb{Z}_p$	a set $\{0, 1, ..., p-1\}$
mod q	reduce an integer, a vector, or a matrix modulo q componentwisely
mod p	reduce an integer, a vector, or a matrix modulo p componentwisely
$[0, N]$	a set $\{0, 1, ..., N-1\}$
$\lfloor \cdot \rceil$	rounding function, $\lfloor x \rceil$ is the nearest integer to the rational number x, rounding upwards in case of a tie
$\|$	concatenation operator
A^t	the transpose of the matrix A
$\|\cdot\|$	norm operator, $\|\mathbf{v}\|$ is 2-norm of the vector $\mathbf{v}$
$\langle \cdot, \cdot \rangle$	inner product
$<<$	a component-wise left shift operation
$>>$	a component-wise right shift operation
$\oplus$	a component-wise XOR operation
$x \leftarrow D$	sampling x from the distribution D
$x \leftarrow X$	sampling x from the set X uniform randomly
λ	the security parameter
$\mathsf{negl}(\cdot)$	the negligible function with respect to the contents of $\mathsf{negl}(\cdot)$
$\mathcal{HWT}_m(h)$	the uniform distribution over the subset of $\{-1, 0, 1\}^m$ whose elements contain $m - h$ number of zeros
$\mathcal{ZO}_n(\rho)$	the distribution over $\{-1, 0, 1\}^n$ where each component x satisfies $\Pr[x = 1] = \Pr[x = -1] = \rho/2$ and $\Pr[x = 0] = 1 - \rho$
$\mathcal{U}_q$	the uniform distribution over $\mathbb{Z}_q$
$\mathcal{DG}_\sigma$	the discrete Gaussian distribution with the parameter σ

$B_{m,h}$	the subset of $\{-1,0,1\}^m$ of which elements have exactly h number of non-zero components, *i.e.* the set of all possible vectors chosen from $\mathcal{HWT}_m(h)$
$B^{\ell}_{m,h}$	the subset of $\{-1,0,1\}^{m\times\ell}$ of which each column has exactly h number of non-zero components
R	$\mathbb{Z}[X]/(X^n+1)$, a ring of polynomials with integer coefficients modulo X^n+1
R_q	R/qR, a set of ring elements in R modulo q
R_p	R/pR, a set of ring elements in R modulo p
R_2	$R/2R$, a set of ring elements in R modulo 2

2 Security Assumptions and Design Rationale

In this section, we introduce the security assumptions exploited in our schemes, and then explain our design rationale for proposed schemes.

2.1 Learning with Errors and Learning with Roundings

Since Regev [40] introduced the LWE problem, a lot of cryptosystems based on this problem have been proposed relying on its versatility. For an n-dimensional vector $\mathbf{s} \in \mathbb{Z}^n$ and an error distribution χ over $\mathbb{Z}$, the LWE distribution $A^{\mathsf{LWE}}_{n,q,\chi}(\mathbf{s})$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is obtained by choosing a vector $\mathbf{a}$ uniformly and randomly from $\mathbb{Z}_q^n$ and an error e from χ, and outputting

$$(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q.$$

The search LWE problem is to find $\mathbf{s} \in \mathbb{Z}_q$ for given arbitrarily many independent samples $(\mathbf{a}_i, b_i)$ from $A^{\mathsf{LWE}}_{n,q,\chi}(\mathbf{s})$. The decision LWE, denoted by $\mathsf{LWE}_{n,q,\chi}(\mathcal{D})$, aims to distinguish the distribution $A^{\mathsf{LWE}}_{n,q,\chi}(\mathbf{s})$ from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ with non-negligible advantage, for a fixed $\mathbf{s} \leftarrow \mathcal{D}$. When the number of samples are limited by m, we denote the problem by $\mathsf{LWE}_{n,m,q,\chi}(\mathcal{D})$.

In this paper, we only consider the discrete Gaussian $\chi = \mathcal{D}G_{\alpha q}$ as an error distribution where α is the error rate in $(0,1)$, so α will substitute the distribution χ in description of LWE problem, say $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{D})$. The LWE problem is self-reducible, so we usually omit the key distribution $\mathcal{D}$ when it is a uniform distribution over $\mathbb{Z}_q^n$.

The hardness of the decision LWE problem is guaranteed by the worst case hardness of the standard lattice problems: the decision version of the *shortest vector problem* (GapSVP), and the *shortest independent vectors problem* (SIVP). After Regev [40] presented the quantum reduction from those lattice problems to the LWE problem, Peikert et al. [15, 37] improved the reduction to a classical version for significantly worse parameters; the dimension should be of the size of $\omega(n \log q)$. In this case, note that the reduction holds only for the GapSVP, not SIVP.

After the works on the connection between the LWE problem and some lattice problems, some variants of LWE, of which the secret distributions are modified from the uniform distribution, were proposed. In [15], Brakerski et al. proved that the LWE problem with binary secret is at least as hard as the original LWE problem. Following the approach of [15], Cheon et al. [17] proved the hardness of the LWE problem with sparse secret, *i.e.*, the number of non-zero components of the secret vector is a constant.

As results of Theorem 4 in [17], the hardness of the LWE problems with (sparse) small secret, $\mathsf{LWE}_{n,m,q,\beta}(\mathcal{HWT}_n(h))$ and $\mathsf{LWE}_{n,m,q,\beta}(\mathcal{ZO}_n(\rho))$ for $0 < \beta < 1$, are guaranteed by the following theorem.

Theorem 1. *(Informal) For positive integers m, n, k, q, h, $0 < \alpha, \beta < 1$ and $0 < \rho < 1$, following statements hold:*

1. *If* $\log({}_nC_h) + h > k \log q$ *and* $\beta > \alpha\sqrt{10h}$*, then the* $\mathsf{LWE}_{n,m,q,\beta}(\mathcal{HWT}_n(h))$ *problem is at least as hard as the* $\mathsf{LWE}_{k,m,q,\alpha}$ *problem.*
2. *If* $\left((1-\rho)\log\left(\frac{1}{1-\rho}\right) + \rho\log\frac{2}{\rho}\right) n > k\log q$ *and* $\beta > \alpha\sqrt{10n}$*, the* $\mathsf{LWE}_{n,m,q,\beta}(\mathcal{ZO}_n(\rho))$ *problem is at least as hard as the* $\mathsf{LWE}_{k,m,q,\alpha}$ *problem.*

In [14, 38, 39], to pack a string of plaintexts in a ciphertext, LWE with single secret was generalized to LWE with multiple secrets. An instance of multi-secret LWE is $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s}_1\rangle + \mathbf{e}_1, ..., \langle \mathbf{a}, \mathbf{s}_k\rangle + \mathbf{e}_k)$ where $\mathbf{s}_1, ..., \mathbf{s}_k$ are secret vectors and $\mathbf{e}_1, ..., \mathbf{e}_k$ are independently chosen error vectors. Using the hybrid argument, multi-secret LWE is proved to be at least as hard as LWE with single secret.

The LWR problem was firstly introduced by Banerjee et al. [10] to improve the efficiency of pseudorandom generator based on the LWE problem. Unlikely to the LWE problem, errors in the LWR problem are deterministic so that the problem is so-called a "derandomized" version of the LWE problem. To hide secret information, the LWR problem uses a rounding by a modulus p instead of inserting errors. Then, the deterministic error is created by scaling down from $\mathbb{Z}_q$ to $\mathbb{Z}_p$.

For an n-dimensional vector $\mathbf{s}$ over $\mathbb{Z}_q$, the LWR distribution $A^{\mathsf{LWR}}_{n,q,p}(\mathbf{s})$ over $\mathbb{Z}_q^n \times \mathbb{Z}_p$ is obtained by choosing a vector $\mathbf{a}$ from $\mathbb{Z}_q^n$ uniform randomly, and returning

$$\left(\mathbf{a}, \left\lfloor \frac{p}{q} \cdot (\langle \mathbf{a}, \mathbf{s}\rangle \bmod q) \right\rceil\right) \in \mathbb{Z}_q^n \times \mathbb{Z}_p.$$

As in the LWE problem, $A^{\mathsf{LWR}}_{n,m,q,p}(\mathbf{s})$ denotes the distribution of m samples from $A^{\mathsf{LWR}}_{n,q,p}(\mathbf{s})$; that is contained in $\mathbb{Z}_q^{m\times n} \times \mathbb{Z}_p^m$. The search LWR problem are defined respectively as finding secret $\mathbf{s}$ just as same as the search version of LWE problem. In contrary, the decision $\mathsf{LWR}_{n,m,q,p}(\mathcal{D})$ problem aims to distinguish the distribution $A^{\mathsf{LWR}}_{n,m,q,p}(\mathbf{s})$ from the uniform distribution over $\mathbb{Z}_q^{m\times n} \times \mathbb{Z}_p^m$ with m instances for a fixed $\mathbf{s} \leftarrow \mathcal{D}$.

In [10], Banerjee et al. proved that there is an efficient reduction from the LWE problem to the LWR problem for a modulus q of super-polynomial size. Later, the follow-up works by Alwen et al. [8] and Bogdanov et al. [12] improved the reduction by eliminating the restriction on modulus size and adding a condition of the bound of the number of samples. In particular, the reduction by Bogdanov et al. works when $2mBp/q$ is a constant, where B is a bound of errors in the LWE problem, m is the number of samples in both problems, and p is the rounding modulus in the LWR problem. That is, the rounding modulus p is proportional to $1/m$ for fixed q and B. Since the reduction from LWE to LWR is independent of the secret distribution, the hardness of $\mathsf{LWR}_{n,m,q,p}(\mathcal{HWT}_n(h))$ and $\mathsf{LWR}_{n,m,q,p}(\mathcal{ZO}_n(\rho))$ is obtained from that of the LWE problems with corresponding secret distributions.

2.2 Ring variants of LWE and LWR

In [33], Lyubashevsky et al. deal with the LWE problem over rings, namely ring-LWE. For positive integers n and q, and an irreducible polynomial $g(x) \in \mathbb{Z}[x]$ of degree n, we define the ring $R = \mathbb{Z}[x]/(g(x))$ and its quotient ring modulo q, $R_q = \mathbb{Z}_q[x]/(g(x))$. We denote the polynomial multiplication of a and b in R and R_q by $a * b$. The ring-LWE problem is to distinguish between the uniform distribution and the distribution of $(a, a * s + e) \in R_q^2$ where a is uniform randomly chosen polynomial, e is chosen from an error distribution, and s is a secret polynomial.

Due to the efficiency and compactness of ring-LWE, many lattice-based cryptosystems are constructed as *ring*-LWE *based*, rather than LWE-based. As with the LWE problem, the ring-LWE problem over the ring R is at least as hard as the search version of approximate SVP over the ideal lattices of R, in the sense of quantum reduction.

The ring variant of LWR is introduced in [10, 12] as an analogue of LWR. In the ring-LWR problem, the vectors chosen from $\mathbb{Z}_q^n$ are substituted by polynomials in R_q, *i.e.*, the ring-LWR instance for a secret polynomial $s \in R_q$ is

$$\left(a, \left\lfloor \frac{p}{q} \cdot a * s \right\rceil\right) \in R_q \times R_p$$

where $\lfloor (p/q) \cdot a * s \rceil$ is obtained by applying the rounding function to each coefficient of $(p/q) \cdot a * s$. The search and decision ring-LWR problems are defined the same way as the LWR problem, but over rings.

In [10], Banerjee et al. proved that decision ring-LWR is at least as hard as decision ring-LWE for sufficiently large modulus. Later, reduction from search ring-LWE to search ring-LWR was constructed in overall scope of the modulus [10] when the number of samples is bounded.

2.3 Design Rationale

Our first IND-CPA secure PKE scheme simply relies on the hardness assumption of the LWE and LWR problems with particular secret distributions. As explained in Section 2, it is shown that LWE with small secret is still hard to solve if the min-entropy of the secret distribution is sufficiently large. Moreover, the LWR problem is somewhat equivalent to LWE unless we overuse the same secrets to generate samples due to the reduction in the recent work [12]. All these aspects lead us to design the primitives named after "Lizard", of which basic goal is to obtain the fastest implementation for encryption and decryption among the lattice-based schemes while maintaining the weaker assumptions, and make the ciphertext sizes smaller in factor $\log q / \log p$.

To give an intuition for the basic algorithms, we describe our Lizard in the case of bit encryption as follows. In the key generation step, we first sample a secret vector $\mathbf{s} \in \{-1, 0, 1\}^n$, a random matrix $A \in \mathbb{Z}_q^{m \times n}$, and an error vector $\mathbf{e} \leftarrow \mathcal{DG}_\sigma^m$ of which components are expected to be small. Then output the secret key $\mathsf{sk} \leftarrow \mathbf{s}$, and public key $\mathsf{pk} \leftarrow (A, \mathbf{b})$ where $\mathbf{b} = A\mathbf{s} + \mathbf{e} \in \mathbb{Z}_q^m$. Hence, the public key is an instance of LWE with the secret vector $\mathbf{s}$. In the encryption step, we sample a sparse signed binary vector $\mathbf{r} \leftarrow \mathcal{HWT}_m(h_r)$ with low Hamming weight $h_r \approx \mathcal{O}(\lambda)$, which is an ephemeral secret of the algorithm. The re-randomization process after calculating $(A^t\mathbf{r}, \mathbf{b}^t\mathbf{r})$ is to adapt the ordinary rounding procedure from the modulus q to lower modulus p, without adding auxiliary noises. The resulting ciphertext for $m \in \{0, 1\}$ is

$$\mathbf{c} \leftarrow (\lfloor (p/q) \cdot A^t\mathbf{r} \rceil,\ \lfloor (p/2) \cdot m + (p/q) \cdot \mathbf{b}^t\mathbf{r} \rceil) \in \mathbb{Z}_p^{n+1},$$

where $\lfloor \cdot \rceil$ denotes the component-wise rounding of entries to the closest integers, rounding upwards in case of a tie. If both $p < q$ are power-of-twos, the rounding procedure could be reduced to the two simple steps: addition of $q/2p$ and the bitwise shift operation. That is, we "cut off" the least significant bits of each component of the vector $(\mathbf{r}^t A, \mathbf{r}^t\mathbf{b})$ to return a ciphertext.

The advantages of Lizard can be analyzed (See Section 3.3 in [18]), but we would like to make simple remarks here. Since the recent LWE attack for using the sparse secrets emerges [2], our parameter has been loosened than previous. However, since we use the sparse signed binary secrets or signed binary secrets, we can obtain the record-breaking encryption and decryption speeds which are faster than those of NTRU respectively, despite the weaker assumption for the security. Using LWR in the encryption phase is better than using LWE because it does not require noise sampling, which results some efficiency, and we have smaller ciphertexts since the factor $\log q$ in the ciphertext size can be reduced to $\log p$. For the usage that requires smaller public key, we can provide our encryption scheme simply replacing the public key with small seed, or turn to the ring version of our scheme called RLizard.

The RLizard CPA secure PKE scheme provides a trade-off between space-efficiency and security, which is of independent interest. In RLizard, a public key is parsed into two structured square matrices modulo q which represent polynomials in R_q, respectively. Hence, the public key size is reduced from $m(n+\ell)\log q$ to $2n\log q$ compared to Lizard. Let $\mathsf{pk} = (a, b)$. The resulting ciphertext for $m \in R_2$ is

$$c \leftarrow (\lfloor (p/q) \cdot a * r \rceil,\ \lfloor (p/2) \cdot m + (p/q) \cdot b * r \rceil) \in R_p^2,$$

where r is an ephemeral secret in the encryption procedure which is a sparse signed binary polynomial, and $*$ denotes multiplication in R_q. It can be seen that all the operations in encryption are just the same with those in Lizard except that multiplications and additions are held in the polynomial space R_q.

2.4 Proposed Schemes

We first propose IND-CPA secure encryption schemes: Lizard and RLizard. To avoid an abuse of notations, we call them "Lizard.CPA" and "RLizard.CPA" through the whole document. We convert Lizard.CPA (*resp.* RLizard.CPA) into an IND-CCA2 Key Encapsulation Mechanism (KEM) Lizard.KEM (*resp.* RLizard.KEM) using a KEM variant of Fujisaki-Okamoto transformation [24, 20, 26]. We also suggest Lizard.CCA (*resp.* RLizard.CCA) using the same transformation, combining it with a One-Time Pad (OTP).

3 Algorithm Specifications

3.1 Symmetric primitives

In our IND-CCA2 schemes, we need to generate (pseudo-)random numbers and hash outputs. We use the pseudorandom generator `randombytes` to generate a random bit string of an arbitrary length, which is recommended to use by NIST. We instantiate all the hash functions in this proposal with `TupleHash256` considering two main factors: the flexibility in input and output lengths, and the long-term security which comes close to that of AES256.

More precisely, we use three hash functions G, H, and H' to achieve the IND-CCA2 security of proposed schemes. The functions G and H' are exactly the `TupleHash256` with proper input and output lengths, while the function H is not: the output of H is generated from the output of `TupleHash256` to be spread following a particular distribution. We specified the exact algorithm to obtain an output of H using `TupleHash256` in Section 6.

3.2 IND-CPA Public Key Encryption Schemes

In this section and through the whole document, we suggest two kinds of IND-CPA secure PKE schemes called Lizard.CPA and RLizard.CPA. The Lizard.CPA and RLizard.CPA PKEs contain three algorithms in each: a key generation Lizard.CPA.KeyGen, encryption Lizard.CPA.Enc and a decryption Lizard.CPA.Dec in the former one, and a key generation RLizard.CPA.KeyGen, encryption RLizard.CPA.Enc and a decryption RLizard.CPA.Dec in the latter one. We assume that certain conditions for inputs hold for the specifications of algorithms, e.g. the public and secret keys are valid, which means they are correctly in their form of the corresponding key types.

3.2.1 The Lizard.CPA Encryption Scheme

For positive integers m, n, ℓ, p, q and h_r such that $h_r < m$ and $2|p|q$, and $0 < \rho, \alpha < 1$, let $params \leftarrow (m, n, q, p, \ell, \rho, h_r, \alpha)$ through all the algorithms here.

Lizard.CPA.KeyGen.

Input: The set of public parameters $params$.

Output: A key pair consisting of the private key $S \in \{-1, 0, 1\}^{n\times\ell}$ and the public key $(A\|B) \in \mathbb{Z}_q^{m\times(n+\ell)}$.

Operation:

1. Generate a random matrix $A \leftarrow \mathbb{Z}_q^{m\times n}$.
2. Set a secret matrix $S := (\mathbf{s}_0\| \cdots \|\mathbf{s}_{\ell-1})$ by sampling each $\mathbf{s}_i$ independently from the distribution $\mathcal{Z}O_n(\rho)$.
3. For $0 \le i \le m-1$ and $0 \le j \le \ell-1$, sample an integer $E_{ij} \leftarrow \mathcal{D}G_{\alpha q}$, and then set $E = (E_{ij}) \in \mathbb{Z}_q^{m\times\ell}$.
4. Compute $B := -AS + E \in \mathbb{Z}_q^{m\times\ell}$.
5. Output the public key $\mathsf{pk} := (A\|B) \in \mathbb{Z}_q^{m\times(n+\ell)}$ and the private key $\mathsf{sk} := S \in \{-1, 0, 1\}^{n\times\ell}$.

Lizard.CPA.Enc.

Inputs: The set of public parameters $params$, the public key $\mathsf{pk} = (A\|B) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^{m\times\ell}$, and the message $\mathbf{M} \in \{0, 1\}^\ell$.

Output: The ciphertext $\mathbf{c} = (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_p^n \times \mathbb{Z}_p^\ell$.

Operation:

1. Generate an m dimensional vector $\mathbf{r} \in B_{m,h_r}$ from the distribution $\mathcal{H}WT_m(h_r)$.
2. Compute $\mathbf{a} := \lfloor (p/q) \cdot A^t\mathbf{r} \rceil \in \mathbb{Z}_p^n$ and $\mathbf{b} := \lfloor (p/q) \cdot ((q/2) \cdot \mathbf{M} + B^t\mathbf{r}) \rceil \in \mathbb{Z}_p^\ell$.
3. Output the ciphertext $\mathbf{c} := (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_p^n \times \mathbb{Z}_p^\ell$.

Lizard.CPA.Dec.

Inputs: The set of public parameters $params$, the secret key $\mathsf{sk} = S \in \{-1, 0, 1\}^{n\times\ell}$ and the ciphertext $\mathbf{c} = (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}_p^n \times \mathbb{Z}_p^\ell$.

Output: The message $\mathbf{M} \in \{0, 1\}^\ell$.

Operation:

1. Parse the ciphertext $\mathbf{c} = (\mathbf{a}, \mathbf{b})$.
2. Compute $\mathbf{M} = \lfloor (2/p) \cdot (\mathbf{b} + S^t\mathbf{a}) \rceil \in \mathbb{Z}_2^\ell$.
3. Output the message $\mathbf{M}$.

3.2.2 The RLizard.CPA Encryption Scheme

For positive integers n, p, q, h_s and h_r such that $h_s, h_r < n$ and $2|p|q$, and $0 < \alpha < 1$, let $params \leftarrow (n, q, p, h_s, h_r, \alpha)$ through all the algorithms here. We denote $R = \mathbb{Z}[x]/(x^n + 1)$ and $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. We identify the polynomial $a = \sum_{i=0}^{n-1} a_i x^i \in R$ $(resp.\ R_q)$ with the vector $\mathbf{a} = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}^n$ $(resp.\ \mathbb{Z}_q^n)$. Therefore, for a polynomial $a \in R$ $(resp.\ R_q)$ and any distribution $\mathcal{D}$ over $\mathbb{Z}^n$ $(resp.\ \mathbb{Z}_q^n)$, $a \leftarrow \mathcal{D}$ means sampling the vector $\mathbf{a}$ following the distribution $\mathcal{D}$ and then identifying the vector with its corresponding polynomial a.

RLizard.CPA.KeyGen.

Input: The set of public parameters $params$.

Output: A key pair containing the private key $s \in R$ and the public key $(a, b) \in R_q^2$.

Operation:
1. Generate a random polynomial $a \leftarrow R_q$.
2. Set a secret polynomial s by sampling it from the distribution $\mathcal{HWT}_n(h_s)$.
3. For $0 \leq i \leq n-1$, sample an integer $e_i \leftarrow \mathcal{DG}_{\alpha q}$, and then set $e = \sum_{i=0}^{n-1} e_i X^i \in R_q$.
4. Compute $b := -a * s + e \in R_q$.
5. Output the public key $\mathsf{pk} := (a, b) \in R_q^2$ and the secret key $\mathsf{sk} := s \in R$.

RLizard.CPA.Enc.

Inputs: The set of public parameters $params$, the public key $(a, b) \in R_q^2$, and the message polynomial $M \in R_2$.

Output: The ciphertext $\mathbf{c} = (c_1, c_2) \in R_p^2$.

Operation:
1. Generate a polynomial $r \in R_q$ by sampling it from the distribution $\mathcal{HWT}_n(h_r)$.
2. Set $c_1' := a * r$, and $c_2' := b * r$ in R_q.
3. Compute $c_1 := \lfloor (p/q) \cdot c_1' \rceil \in R_p$ and $c_2 := \lfloor (p/q) \cdot ((q/2) \cdot M + c_2') \rceil \in R_p$.
4. Output the ciphertext $\mathbf{c} := (c_1, c_2)$.

RLizard.CPA.Dec.

Inputs: The set of public parameters $params$, the secret key $\mathsf{sk} = s \in R$, and the ciphertext $\mathbf{c} = (c_1, c_2) \in R_p^2$.

Output: The message $m \in R_2$.

Operation:
1. Parse the ciphertext $\mathbf{c} = (c_1, c_2)$.
2. Compute $M := \lfloor (2/p) \cdot (c_2 + c_1 * s) \rceil \in R_2$.
4. Output the message m.

3.3 IND-CCA2 Key Encapsulation Mechanisms

In this section, we suggest two kinds of IND-CCA2 KEM, Lizard.KEM and RLizard.KEM, which are derived by CCA KEM conversions [26] of Lizard.CPA and RLizard.CPA, respectively.

3.3.1 Overview

Recently, Hofheinz et al. [26] suggested a modular toolkit of FO transformations [24, 20, 43], which turns an arbitrary weakly (i.e., IND-CPA) secure PKE into a strongly (i.e., IND-CCA2) secure key encapsulation in the (quantum) random oracle model. The transformation has certain merits since it is robust against schemes with nonzero decryption failure probability while the

others are not. We utilize their conversion technique in quantum random oracle model for our CPA-secure Lizard and RLizard to achieve the IND-CCA2 KEMs.

Basically, the symmetric primitives required in the IND-CCA2 secure Lizard/RLizard KEMs are the same as in the IND-CCA2 secure Lizard PKE. That is, we use three hash functions G, H, H', where G and H' output a d-bit string where d denotes the bit-length of messages of the CCA schemes and H outputs m-bit string(s) with hamming weight h_r, and the OTP here. The one thing changed in Lizard.CPA to obtain Lizard.KEM is that we transform the message vector of the length ℓ to the matrix of the size $\ell_1 \times \ell_2$ for some ℓ_1 and ℓ_2 such that $\ell_1 \cdot \ell_2 = \ell$, and use the parameters (ℓ_1, ℓ_2) instead of ℓ. Our Lizard.CPA can be re-written in the matrix form as follows:

- The key pair are generated normally as

$$\mathsf{pk} \leftarrow (A \| B) \in \mathbb{Z}_q^{m \times (n+\ell_1)}, \ \mathsf{sk} \leftarrow S \in \{-1, 0, 1\}^{n \times \ell_1}$$

- For a message $M \in \{0,1\}^{\ell_1 \times \ell_2}$, we first generate an ephemeral secret as a matrix

$$\begin{aligned} &R \leftarrow \mathcal{HWT}_m(h_r)^{\ell_2} \in \{-1, 0, 1\}^{m \times \ell_2}, \text{ and calculate} \\ &C \leftarrow (\lfloor (p/q) \cdot A^t R \rceil, \lfloor (p/2) \cdot M + (p/q) \cdot B^t R \rceil) \in \mathbb{Z}_p^{n \times \ell_2} \times \mathbb{Z}_p^{\ell_1 \times \ell_2}, \end{aligned}$$

  where $\lfloor \cdot \rceil$ denotes componentwise rounding for whole matrix.

We use this form for Lizard.CPA to make the public key size and the ciphertext size somewhat balanced. Actually, the public key size is reduced by a factor ℓ_2, and the ciphertext size grows from $(n+\ell)\log p$ to $(n \cdot \ell_2 + \ell)\log p$ in this matrix form of Lizard.CPA. On the other hand, RLizard.KEM is obtained by applying the conversion technique directly to RLizard.CPA.

3.3.2 The Lizard.KEM Scheme

For positive integers m, n, ℓ_1, ℓ_2, ℓ, d, p, q and h_r such that $h_r < m$, $\ell = \ell_1 \cdot \ell_2$, and $2|p|q$, $0 < \rho, \alpha < 1$, and the hash functions $G : \{0,1\}^* \to \{0,1\}^d$, $H : \{0,1\}^* \to B_{m,h_r}^{\ell_2}$ and $H' : \{0,1\}^* \to \{0,1\}^\ell$, let $params \leftarrow (m, n, q, p, \ell_1, \ell_2, \ell, d, \rho, h_r, \alpha, G, H, H')$ through all the algorithms here.

Lizard.KEM.KeyGen.

Input: The set of parameters $params$.

Output: A key pair containing the private key $(S, T) \in \{-1, 0, 1\}^{n \times \ell_1} \times \{0,1\}^{\ell_1 \times \ell_2}$ and the public key $(A \| B) \in \mathbb{Z}_q^{m \times (n+\ell_1)}$.

Operation:

1. Generate a random matrix $A \leftarrow \mathbb{Z}_q^{m \times n}$.
2. Set a secret matrix $S := (\mathbf{s}_0 \| \cdots \| \mathbf{s}_{\ell_1 - 1})$ by sampling each $\mathbf{s}_i$ independently from the distribution $\mathcal{ZO}_n(1/2)$.
3. Generate a random matrix $T \leftarrow \{0,1\}^{\ell_1 \times \ell_2}$.
4. For $0 \le i \le m-1$ and $0 \le j \le \ell_1 - 1$, sample an integer $E_{ij} \leftarrow \mathcal{DG}_{\alpha q}$, and then set $E = (E_{ij}) \in \mathbb{Z}_q^{m \times \ell_1}$.
5. Compute $B := -AS + E \in \mathbb{Z}_q^{m \times \ell_1}$.
6. Output the public key $\mathsf{pk} := (A \| B) \in \mathbb{Z}_q^{m \times (n+\ell_1)}$ and the secret key $\mathsf{sk} := (S, T) \in \{-1, 0, 1\}^{n \times \ell_1} \times \{0,1\}^{\ell_1 \times \ell_2}$.

Lizard.KEM.Encaps.

Inputs: The set of public parameters *params*, public key $\mathsf{pk} = (A\|B) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^{m\times \ell_1}$.

Output: The ciphertext $C = (C_1, C_2, \mathbf{d}) \in \mathbb{Z}_p^{n\times \ell_2} \times \mathbb{Z}_p^{\ell_1\times \ell_2} \times \{0,1\}^{\ell}$ and the shared key $\mathbf{K} \in \{0,1\}^d$.

Operation:
1. Generate a random matrix $M \in \{0,1\}^{\ell_1\times \ell_2}$.
2. Compute the matrix $R := H(M)$ and the vector $\mathbf{d} := H'(M)$.
3. Compute $C_1 := \lfloor (p/q)\cdot A^t R \rceil \in \mathbb{Z}_p^{n\times \ell_2}$ and $C_2 := \lfloor (p/q)\cdot ((q/2)\cdot M + B^t R)\rceil \in \mathbb{Z}_p^{\ell_1\times \ell_2}$.
4. Compute $\mathbf{K} := G(C_1, C_2, \mathbf{d}, M)$, and output the pair $(C = (C_1, C_2, \mathbf{d}), \mathbf{K})$.

Lizard.KEM.Decaps.

Inputs: The set of public parameters *params*, the public key $\mathsf{pk} = (A\|B) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^{m\times \ell_1}$, the secret key $\mathsf{sk} = (S,T) \in \{-1,0,1\}^{n\times \ell_1} \times \{0,1\}^{\ell_1\times \ell_2}$, and the ciphertext $C = (C_1, C_2, \mathbf{d}) \in \mathbb{Z}_p^{n\times \ell_2} \times \mathbb{Z}_p^{\ell_1\times \ell_2} \times \{0,1\}^{\ell}$.

Output: The shared key $\mathbf{K} \in \{0,1\}^d$.

Operation:
1. Parse the ciphertext $C := (C_1, C_2, \mathbf{d})$.
2. Compute $M' := \lfloor (2/p)\cdot (C_2 + S^t C_1)\rceil \in \mathbb{Z}_2^{\ell_1\times \ell_2}$.
3. Compute $R' := H(M')$ and $\mathbf{d}' := H'(M')$.
4. Compute $C_1' := \lfloor (p/q)\cdot A^t R'\rceil \in \mathbb{Z}_p^{n\times \ell_2}$ and $C_2' := \lfloor (p/q)\cdot ((q/2)\cdot M' + B^t R')\rceil \in \mathbb{Z}_p^{\ell_1\times \ell_2}$, and set $C' := (C_1', C_2', \mathbf{d}')$.
5. If $C \neq C'$, then output $\mathbf{K} := G(C_1, C_2, \mathbf{d}, T)$.
6. Else, output the shared key $\mathbf{K} := G(C_1, C_2, \mathbf{d}, M')$.

3.3.3 The RLizard.KEM Scheme

For positive integers n, d, p, q, h_r, and h_s such that $h_r, h_s < n$ and $2|p|q$, $0 < \alpha < 1$, and the hash functions $G : R_p \times R_p \times \{0,1\}^d \times R_2 \to \{0,1\}^d$, $H : R_2 \to B_{n,h_r}$ and $H' : R_2 \to \{0,1\}^n$, let $params \leftarrow (n, q, p, d, h_s, h_r, \alpha, G, H, H')$ through all the algorithms here.

RLizard.KEM.KeyGen.

Input: The set of public parameters *params*.

Output: A key pair containing the private key $(s,t) \in R \times R_2$ and the public key $(a,b) \in R_q^2$.

Operation:
1. Generate a random polynomial $a \leftarrow R_q$.
2. Set a secret polynomial $s \leftarrow \mathcal{HWT}_n(h_s)$.
3. Generate a random vector $\mathbf{t} \leftarrow \{0,1\}^n$ and identify it with the polynomial $t \in R_2$.
4. For $0 \le i \le n-1$, sample an integer $e_i \leftarrow \mathcal{DG}_{\alpha q}$, and then set $e = \sum_{i=0}^{n-1} e_i X^i \in R_q$.
5. Compute $b := -a * s + e \in R_q$.
6. Output the public key $\mathsf{pk} := (a,b) \in R_q^2$ and the secret key $\mathsf{sk} := (s,t) \in R \times R_2$.

RLizard.KEM.Encaps.

Inputs: The set of public parameters *params*, the public key $\mathsf{pk} := (a, b) \in R_q^2$.

Output: The ciphertext $\mathbf{c} := (c_1, c_2, \mathbf{d}) \in R_p \times R_p \times \{0,1\}^n$ and the shared key $\mathbf{K} \in \{0,1\}^d$.

Operation:
1. Generate a polynomial $\delta \leftarrow R_2$.
2. Compute $r := H(\delta)$ and $\mathbf{d} := H'(\delta)$.
3. Compute $c_1 := \lfloor (p/q) \cdot a * r \rceil \in R_p$ and $c_2 := \lfloor (p/q) \cdot ((q/2) \cdot \delta + b * r) \rceil \in R_p$.
4. Compute $\mathbf{K} := G(c_1, c_2, \mathbf{d}, \delta)$.
5. output $(c_1, c_2, \mathbf{d}, \mathbf{K})$.

RLizard.KEM.Decaps.

Inputs: The set of public parameters *params*, the public key $\mathsf{pk} := (a, b) \in R_q^2$, the secret key $(s, t) \in R \times R$, and the ciphertext $\mathbf{c} := (c_1, c_2, \mathbf{d}) \in R_p \times R_p \times \{0,1\}^n$.

Output: The shared key $\mathbf{K} \in \{0,1\}^d$.

Operation:
1. Parse the ciphertext $\mathbf{c} := (c_1, c_2, \mathbf{d})$.
2. Compute $\delta' := \lfloor (2/p) \cdot (c_2 + s * c_1) \rceil \in R_2$.
3. Compute $r' := H(\delta')$ and $\mathbf{d}' := H'(\delta')$.
4. Compute $a' := \lfloor (p/q) \cdot a * r' \rceil \in R_p$ and $b' := \lfloor (p/q) \cdot ((q/2) \cdot \delta' + b * r') \rceil \in R_p$, and set $\mathbf{c}' := (a', b', \mathbf{d}')$.
5. If $\mathbf{c} \neq \mathbf{c}'$, then output $\mathbf{K} = G(c_1, c_2, \mathbf{d}, t)$.
6. Else, output the shared key $\mathbf{K} = G(c_1, c_2, \mathbf{d}, \delta')$.

3.4 IND-CCA2 Public Key Encryption Schemes

In this section, we suggest two kinds of IND-CCA2 public key encryption schemes. We apply a simple conversion for our KEMs to obtain these IND-CCA2 PKEs. The conversion modifies the encapsulation algorithm simply by appending OTP encryption of a message in $\mathbb{Z}_2^d$ to the key value of the KEM. Our IND-CCA2 PKE Lizard and RLizard are specified as Lizard.CCA and RLizard.CCA, respectively.

3.4.1 The Lizard.CCA Scheme

For positive integers m, n, ℓ, d, p, q and h_r such that $h_r < m$ and $2|p|q$, $0 < \rho, \alpha < 1$, and the hash functions $G : \{0,1\}^* \to \{0,1\}^d$, $H : \{0,1\}^* \to B_{m,h_r}$ and $H' : \{0,1\}^* \to \{0,1\}^\ell$, let $params \leftarrow (m, n, q, p, \ell, d, \rho, h_r, \alpha, G, H, H')$ through all the algorithms here.

Lizard.CCA.KeyGen.

Input: The set of public parameters *params*.

Output: A key pair containing the private key $S \in \{-1, 0, 1\}^{n \times \ell}$ and the public key $(A \| B) \in \mathbb{Z}_q^{m \times (n+\ell)}$.

Operation:

1. Generate a random matrix $A \leftarrow \mathbb{Z}_q^{m \times n}$.
2. Set a secret matrix $S := (\mathbf{s}_0 \| \cdots \| \mathbf{s}_{\ell-1})$ by sampling each $\mathbf{s}_i$ independently from the distribution $\mathcal{Z}O_n(\rho)$.
3. For $0 \leq i \leq m-1$ and $0 \leq j \leq \ell - 1$, sample an integer $E_{ij} \leftarrow \mathcal{D}G_{\alpha q}$, and then set $E = (E_{ij}) \in \mathbb{Z}_q^{m \times \ell}$.
4. Compute $B := -AS + E \in \mathbb{Z}_q^{m \times \ell}$.
5. Output the public key $\mathsf{pk} := (A \| B) \in \mathbb{Z}_q^{m \times (n+\ell)}$ and the private key $\mathsf{sk} := S \in \{-1, 0, 1\}^{n \times \ell}$.

Lizard.CCA.Enc.

Input: The set of public parameters *params*, the public key $\mathsf{pk} = (A \| B) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^{m \times \ell}$, and the message $\mathbf{M} \in \{0,1\}^d$.

Output: The ciphertext $\mathbf{c} = (\mathbf{c}_1, (\mathbf{a}, \mathbf{b}), \mathbf{c}_3) \in \{0,1\}^d \times \mathbb{Z}_p^{n+\ell} \times \{0,1\}^\ell$.

Operation:

1. Generate a random vector $\boldsymbol{\delta} \leftarrow \{0,1\}^\ell$.
2. Set $\mathbf{c}_1 := \mathbf{M} \oplus G(\boldsymbol{\delta}) \in \mathbb{Z}_2^d$ and $\mathbf{c}_3 := H'(\boldsymbol{\delta})$.
3. Set $\mathbf{r} := H(\boldsymbol{\delta}) \in \{-1, 0, 1\}^m$.
4. Compute $\mathbf{a} := \lfloor (p/q) \cdot A^t \mathbf{r} \rceil \in \mathbb{Z}_p^n$ and $\mathbf{b} := \lfloor (p/q) \cdot ((q/2) \cdot \boldsymbol{\delta} + B^t \mathbf{r} \rceil) \in \mathbb{Z}_p^\ell$.
5. Output $\mathbf{c} = (\mathbf{c}_1, (\mathbf{a}, \mathbf{b}), \mathbf{c}_3)$.

Lizard.CCA.Dec.

Input: The set of public parameters *params*, the public key $\mathsf{pk} = (A \| B) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^{m \times \ell}$, the secret key $\mathsf{sk} = S \in \{-1, 0, 1\}^{n \times \ell}$ and the ciphertext $\mathbf{c} = (\mathbf{c}_1, (\mathbf{a}, \mathbf{b}), \mathbf{c}_3) \in \{0,1\}^d \times \mathbb{Z}_p^{n+\ell} \times \{0,1\}^\ell$.

Output: The message $\mathbf{M} \in \{0,1\}^d$.

Operation:

1. Parse the ciphertext $\mathbf{c} := (\mathbf{c}_1, (\mathbf{a}, \mathbf{b}), \mathbf{c}_3)$.
2. Compute $\boldsymbol{\delta} := \lfloor (2/p) \cdot (\mathbf{b} + S^t \mathbf{a}) \rceil \in \mathbb{Z}_2^\ell$.
3. Compute the hash values $G(\boldsymbol{\delta})$ and $H'(\boldsymbol{\delta})$.
4. If $\mathbf{c}_3 \neq H'(\boldsymbol{\delta})$, then abort.
5. Else, compute $\mathbf{r} := H(\boldsymbol{\delta})$, and vectors $\lfloor (p/q) \cdot A^t \mathbf{r} \rceil \in \mathbb{Z}_p^n$ and $\lfloor (p/q) \cdot ((q/2) \cdot \boldsymbol{\delta} + B^t \mathbf{r}) \rceil \in \mathbb{Z}_p^\ell$.
6. If $(\mathbf{a}, \mathbf{b}) \neq (\lfloor (p/q) \cdot A^t \mathbf{r} \rceil, \lfloor (p/q) \cdot ((q/2) \cdot \boldsymbol{\delta} + B^t \mathbf{r} \rceil))$, then abort.
7. Else, compute and output the message $\mathbf{M} := \mathbf{c}_1 \oplus G(\boldsymbol{\delta})$.

3.4.2 The RLizard.CCA Scheme

For positive integers n, p, q, d, h_s and h_r such that $h_s, h_r < n$, and $2|p|q$, and $0 < \alpha < 1$, and the hash functions $G : R_2^* \to \{0,1\}^d$, $H : R_2^* \to B_{n,h_r}$ and $H' : R_2 \to \{0,1\}^n$, let $params \leftarrow (n, q, p, d, h_s, h_r, \alpha, G, H, H')$ through all the algorithms here.

RLizard.CCA.KeyGen.

Input: The set of public parameters $params$.

Output: A key pair containing the private key $s \in R$ and the public key $(a, b) \in R_q^2$.

Operation:

1. Generate a random polynomial $a \leftarrow R_q$.
2. Set a secret polynomial s by sampling it from the distribution $\mathcal{HWT}_n(h_s)$.
3. For $0 \leq i \leq n-1$, sample an integer $e_i \leftarrow \mathcal{DG}_{\alpha q}$, and then set $e = \sum_{i=0}^{n-1} e_i X^i \in R_q$.
4. Compute $b := -a * s + e \in R_q$ where the operations are polynomial operations in R_q.
5. Output the public key $\mathsf{pk} := (a, b) \in R_q^2$ and the secret key $\mathsf{sk} := s \in R$.

RLizard.CCA.Enc.

Input: The set of public parameters $params$, the public key $\mathsf{pk} = (a, b) \in R_q^2$, and the message $\mathbf{m} \in \{0,1\}^d$.

Output: The ciphertext $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3) \in \{0,1\}^d \times R_p^2 \times \{0,1\}^n$.

Operation:

1. Generate a random polynomial $\delta \leftarrow R_2$.
2. Set $\mathbf{c}_1 := \mathbf{m} \oplus G(\delta) \in \{0,1\}^d$ and $\mathbf{c}_3 := H'(\delta)$.
3. Compute $r := H(\delta) \in B_{n,h_r}$.
4. Compute $\mathbf{c}_2 := (\lfloor (p/q) \cdot a * r \rceil, \lfloor (p/q) \cdot ((q/2) \cdot \delta + b * r) \rceil) \in R_p^2$.
5. Output the ciphertext $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$.

RLizard.CCA.Dec.

Input: The set of public parameters $params$, the public key $\mathsf{pk} = (a, b) \in R_q^2$, the secret key $\mathsf{sk} = s \in R$ and the ciphertext $\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3) \in \{0,1\}^d \times R_p^2 \times \{0,1\}^n$.

Output: The message $\mathbf{m} \in \{0,1\}^d$.

Operation:

1. Parse the ciphertext $\mathbf{c} := (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$.
2. Compute $\delta \leftarrow \mathsf{Lizard.CPA.Dec}(\mathsf{sk}, \mathbf{c}_2)$.
3. Compute the hash values $G(\delta)$ and $H'(\delta)$.
4. If $\mathbf{c}_3 \neq H'(\delta)$, then abort.
5. Else, compute $r := H(\delta)$, and polynomials $\lfloor (p/q) \cdot a * r \rceil \in R_p$ and $\lfloor (p/q) \cdot ((q/2) \cdot \delta + b * r) \rceil \in R_p$.
6. If $\mathbf{c}_2 \neq (\lfloor (p/q) \cdot a * r \rceil, \lfloor (p/q) \cdot ((q/2) \cdot \delta + b * r) \rceil)$, then abort.
7. Else, compute and output the message $\mathbf{m} := \mathbf{c}_1 \oplus G(\delta)$.

3.5 Correctness Analyses

The following lemma shows certain condition to make decryption failure probability negligible in λ.

Lemma 1 (Correctness for Lizard.CPA and RLizard.CPA). *The Lizard.CPA scheme works correctly (except for the negligible probability) as long as the following inequality holds for the security parameter λ:*

$$\Pr\left[|\langle \mathbf{e}, \mathbf{r}\rangle + \langle \mathbf{s}, \mathbf{f}\rangle| \geq \frac{q}{4} - \frac{q}{2p}\right] < 2^{-\lambda},$$

where each component e_i of the error vector $\mathbf{e}$ is independently sampled from $\mathcal{DG}_\sigma$, $\mathbf{r} \leftarrow \mathcal{HWT}_m(h_r)$, $\mathbf{s} \leftarrow \mathcal{ZO}_n(\rho)$, and $\mathbf{f} \leftarrow \mathbb{Z}_{q/p}^n$.

Similarly, the RLizard.CPA scheme works correctly if

$$Pr\left[|e * r + s * f| \geq \frac{q}{4} - \frac{q}{2p}\right] < 2^{-\lambda},$$

where each coefficient of $e = \sum_{i=0}^{n-1} e_i' X^i$ is sampled from $\mathcal{DG}_\sigma$, $r \leftarrow \mathcal{HWT}_m(h_r)$, $s \leftarrow \mathcal{HWT}_n(h_s)$, and $f \leftarrow R_{q/p}$.

Proof. Let $\mathbf{r}$ be an m-dimensional vector sampled from $\mathcal{HWT}_m(h_r)$ in our encryption procedure. The output ciphertext is $\mathbf{c} \leftarrow (\mathbf{c}_1 = \lfloor (p/q) \cdot (A^t\mathbf{r}) \rceil, \mathbf{c}_2 = \lfloor (p/q) \cdot ((q/2) \cdot \mathbf{m} + B^t\mathbf{r}) \rceil)$.

Let $\mathbf{f}_1 \leftarrow \mathbf{c}_1' \pmod{q/p} \in \mathbb{Z}_{q/p}^n$ and $\mathbf{f}_2 \leftarrow \mathbf{c}_2' \pmod{q/p} \in \mathbb{Z}_{q/p}^\ell$ be the vectors satisfying $(q/p)\cdot\mathbf{c}_1 = \mathbf{c}_1' - \mathbf{f}_1$ and $(q/p)\cdot(\mathbf{c}_2 - (p/2)\cdot\mathbf{m}) = \mathbf{c}_2' - \mathbf{f}_2$. Note that $\mathbf{f}_1 = A^t\mathbf{r} \pmod{q/p}$ is uniformly and randomly distributed over $\mathbb{Z}_{q/p}^n$ independently from the choice of $\mathbf{r}$, $\mathbf{e}$, and $\mathbf{s}$. Then for any $1 \leq i \leq \ell$, the i-th component of $\mathbf{c}_2 - S^t\mathbf{c}_1 \in \mathbb{Z}_q^\ell$ is

$$\begin{aligned}(\mathbf{c}_2 - S^t\mathbf{c}_1)[i] &= (p/t)\cdot m_i + (p/q)\cdot(\mathbf{c}_2' - S^t\mathbf{c}_1')[i] - (p/q)\cdot(\mathbf{f}_2[i] - \langle \mathbf{s}_i, \mathbf{f}_1\rangle)\\ &= (p/t)\cdot m_i + (p/q)\cdot(\langle \mathbf{e}_i, \mathbf{r}\rangle + \langle \mathbf{s}_i, \mathbf{f}_1\rangle) - (p/q)\cdot\mathbf{f}_2[i]\\ &= (p/t)\cdot m_i + \lfloor (p/q)\cdot(\langle \mathbf{e}_i, \mathbf{r}\rangle + \langle \mathbf{s}_i, \mathbf{f}_1\rangle)\rceil\end{aligned}$$

since $\mathbf{f}_2 = (AS + E)^t\mathbf{r} = S^t\mathbf{f}_1 + E^t\mathbf{r} \pmod{q/p}$. Therefore, the correctness of our scheme is guaranteed if the encryption error is bounded by $p/4$, or equivalently, $|\langle \mathbf{e}_i, \mathbf{r}\rangle + \langle \mathbf{s}_i, \mathbf{f}_1\rangle| < q/4 - q/2p$ with an overwhelming probability.

Same proof holds for the RLizard.CPA scheme. □

Decryption failure probabilities of Lizard.CCA (*resp.* RLizard.CCA) and Lizard.KEM (*resp.* RLizard.KEM) are equal to that of Lizard.CPA (*resp.* RLizard.CPA) :

Lemma 2 ([26]). *If Lizard.CPA is correct with the probability $1 - \epsilon$, then Lizard.CCA and Lizard.KEM are correct except with the probability $1 - \epsilon$ in the (quantum) random oracle model.*

Samely, if RLizard.CPA is correct with the probability $1-\epsilon$, then RLizard.CCA and RLizard.KEM is correct except with the probability $1 - \epsilon$ in the (quantum) random oracle model.

4 Security Analysis and Recommended parameters

4.1 Security Proofs

4.1.1 IND-CPA Security

We first argue that Lizard.CPA is *IND-CPA secure* under the hardness assumptions of the LWE problem and the LWR problem. The following theorem gives an explicit proof of our argument on security.

Theorem 2 (Security). *The PKE scheme Lizard is IND-CPA secure under the hardness assumption of $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{ZO}_n(\rho))$ and $\mathsf{LWR}_{m,n+\ell,q,p}(\mathcal{HWT}_m(h_r))$.*

Proof. An encryption of $\mathbf{M}$ can be generated by adding $(\mathbf{0}, (p/2) \cdot \mathbf{M}) \in \mathbb{Z}_q^n \times \mathbb{Z}_p^\ell$ to an encryption of zero, since $2|p|q$. Hence, it is enough to show that the pair of public information $\mathsf{pk} = (A\|B) \leftarrow \mathsf{Lizard.CPA.KeyGen}(params)$ and encryption of zero $\mathbf{c} \leftarrow \mathsf{Lizard.CPA.Enc}_{\mathsf{pk}}(\mathbf{0})$ is computationally indistinguishable from the uniform distribution over $\mathbb{Z}_q^{m\times(n+\ell)} \times \mathbb{Z}_q^{n+\ell}$ for a parameter set $params \leftarrow \mathsf{Lizard.CPA.Setup}(1^\lambda)$.

- $\mathcal{D}_0 = \{(\mathsf{pk}, \mathbf{c}) : \mathsf{pk} \leftarrow \mathsf{Lizard.CPA.KeyGen}(params), \mathbf{c} \leftarrow \mathsf{Lizard.CPA.Enc}_{\mathsf{pk}}(\mathbf{0})\}$.
- $\mathcal{D}_1 = \{(\mathsf{pk}, \mathbf{c}) : \mathsf{pk} \leftarrow \mathbb{Z}_q^{m\times(n+\ell)}, \mathbf{c} \leftarrow \mathsf{Lizard.CPA.Enc}_{\mathsf{pk}}(\mathbf{0})\}$.
- $\mathcal{D}_2 = \{(\mathsf{pk}, \mathbf{c}) : \mathsf{pk} \leftarrow \mathbb{Z}_q^{m\times(n+\ell)}, \mathbf{c} \leftarrow \mathbb{Z}_p^{n+\ell}\}$.

The public key $\mathsf{pk} = (A\|B) \leftarrow \mathsf{Lizard.CPA.KeyGen}(params)$ is generated by sampling m instances of LWE problem with ℓ independent secret vectors $\mathbf{s}_1, \ldots, \mathbf{s}_\ell \leftarrow \mathcal{ZO}_n(\rho)$. In addition, the multi-secret LWE problem is no easier than ordinary LWE problem as noted in Section 2.1. Hence, distributions $\mathcal{D}_0$ and $\mathcal{D}_1$ are computationally indistinguishable under the $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{ZO}_n(\rho))$ assumption.

Now assume that pk is uniform random over $\mathbb{Z}_q^{m\times(n+\ell)}$. Then pk and $\mathbf{c} \leftarrow \mathsf{Lizard.CPA.Enc}_{\mathsf{pk}}(\mathbf{0})$ together form $(n+\ell)$ instances of the m dimensional LWR problem with secret $\mathbf{r} \leftarrow \mathcal{H}WT_m(h_r)$. Therefore, distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are computationally indistinguishable under the hardness assumption of $\mathsf{LWR}_{m,n+\ell,q,p}(\mathcal{H}WT_m(h_r))$.

As a result, distributions $\mathcal{D}_0$ and $\mathcal{D}_2$ are computationally indistinguishable under the hardness assumption of $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{ZO}_n(\rho))$ and $\mathsf{LWR}_{m,n+\ell,q,p}(\mathcal{H}WT_m(h_r))$, which denotes the IND-CPA security of the PKE scheme. □

As mentioned on Section 2.1, we know that $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{ZO}_n(\rho))$ and $\mathsf{LWR}_{m,n+\ell,q,p}(\mathcal{H}WT_m(h_r))$ both have reductions from the original LWE problem, which is already proven to be hard. Therefore, Lizard.CPA has a strong security ground. In case of RLizard.CPA, by the similarity of the construction, we can prove that RLizard.CPA is IND-CPA under the hardness assumption of the ring-LWE problem and ring-LWR problem with our secret distributions. As far as we know, there is no known reduction from worst case hard problems to ring-LWE problem and ring-LWR problem because we use the sparse small secrets.

4.1.2 IND-CCA2 Security

Since we obtained the proof for INC-CPA security of Lizard.CPA and RLizard.CPA, we can prove the IND-CCA2 security of Lizard.KEM and RLizard.KEM. We argue that Lizard.KEM and RLizard.KEM achieve tight IND-CCA2 security in the random oracle model, and non-tight IND-CCA2 security in the quantum random oracle model. For IND-CCA2 security in ROM, the hash function H' and the hash value $\mathbf{d}$ are not necessary.

Theorem 3. *([26], Theorem 3.2 and 3.3) For any IND-CCA2 adversary $\mathcal{B}$ on Lizard.KEM issuing at most q_D queries to the decryption oracle, q_G queries to the random oracle G, and q_H queries to the random oracle H, there exists an IND-CPA adversary $\mathcal{A}$ on Lizard.CPA such that*

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{CCA}}_{\mathsf{Lizard.KEM}}(\mathcal{B})\\ &\leq q_G \cdot \epsilon + \frac{q_H}{2^{\omega(\log\lambda)}} + \frac{2q_G+1}{t^\ell} + 3 \cdot \mathsf{Adv}^{\mathsf{CPA}}_{\mathsf{Lizard.CPA}}(\mathcal{A})\end{aligned}$$

where λ is a security parameter and ϵ is a decryption failure probability of Lizard.CPA and Lizard.KEM.

Theorem 4. *([26], Theorem 4.4 and 4.5) For any IND-CCA2 quantum adversary $\mathcal{B}$ on Lizard.KEM issuing at most q_D (classical) queries to the decryption oracle, q_G queries to the quantum random oracle G, q_H queries to the quantum random oracle H, and $q_{H'}$ queries to the quantum random oracle H', there exists an IND-CPA quantum adversary $\mathcal{A}$ on Lizard.CPA such that*

$$\mathsf{Adv}^{CCA}_{\mathsf{Lizard.KEM}}(\mathcal{B}) \leq$$
$$(q_H + 2q_{H'})\sqrt{8\epsilon(q_G+1)^2 + (1+2q_G)\sqrt{\mathsf{Adv}^{CPA}_{\mathsf{Lizard.CPA}}(\mathcal{A})}}$$

where ϵ is a decryption failure probability of Lizard.CPA and Lizard.KEM.

Since Theorem 3 and 4 are using Lizard.CPA as an IND-CPA secure block to prove the IND-CCA2 security of Lizard.KEM, we can easily convert them into the theorems using RLizard.CPA to prove RLizard.KEM is IND-CCA2 secure.
From the similarity of Lizard.KEM and RLizard.KEM, since Lizard.CCA and RLizard.CCA are simply appending OTP encryption of a message in $\mathbb{Z}_2^d$ to the key value of the KEM, we can apply Theorem 3 and Theorem 4 with slight modification. Therefore, Lizard.CCA and RLizard.CCA are also IND-CCA2 secure.

4.2 Parameter Selection

In this section, we analyze the parameter conditions to provide conservative security against known attacks. To do that, we survey all known typical attacks against LWE such as exhaustive search, distinguishing attack, embedding attack, BKW attack [3, 4, 21, 28], etc. Since the LWE problems used in our scheme publish a limited number of samples, it suffices to consider the attacks using lattice basis reduction algorithm. We plugged the BKZ lattice basis reduction algorithm [16, 42] in the attacks, which outputs sufficiently short basis of a lattice according to the time complexity. The most powerful strategies for this kind of attacks in our setting are categorized as follows.

- One can reduce the LWE problem to the Short Integer Solution (SIS) problem. The distinguishing attack analyzed in [34, 41] follows this strategy, which is extended to the dual attack.
- Regarding LWE as the Bounded-Distance Decoding (BDD) problem, one can reduce it to unique-SVP (uSVP). The embedding attack analyzed in [5, 32] follows this strategy, which is extended to the primal attack.
- There are various techniques to adapt the above two strategies for the small secret variants of LWE, e.g. the modulus switching [22], the Bai and Galbraith's rescaling technique for the embedding attack [9], and the BKW style combinatorial approach to the dual attack on LWE [2].

Assembling all methods, we concluded that the dual attack with combinatorial apprach [2] and the primal attack revisited in [1] are the best attacks against the LWE instances in our setting.

We recall the strategies for the attacks against decisional LWE in the following subsections. We also observe that there is no difference between LWE and LWR in the attack contexts. Actually, an instance of the LWR problem can be simply translated into an LWE instance. We would adjust the best attacks against LWE to LWR.

Remark 1. We mainly focus on attacks for LWE and LWR rather than ring-LWE and ring-LWR because we believe that the best attacks against RLizard.KEM and RLizard.CCA are the same attacks on standard lattices where the polynomials are seen as matrices. Hence, we additionally considered attacks against $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{HWT}_n(h_s))$ for analysis of ring based schemes.

4.2.1 Known Attacks on LWE and LWR

In this subsection, we analyze the conditions to make the LWE problem secure against the best attacks, and adjust them to the LWR problem. We achieve the required short vector by running the BKZ algorithm for the target lattice: if Λ is a target lattice of dimension n, then the norms of the shortest vectors in the output of the BKZ algorithm is approximately

$$\|\mathbf{b}_1\| = \delta^n \cdot \det(\Lambda)^{1/n},$$

where δ converges to a constant rapidly as n grows. This δ, called *root Hermite factor*, is used to measure the security of lattice problems. In other words, the runtime of the BKZ algorithm to achieve a given root Hermite factor in large dimension (> 200) is determined heuristically by δ. In analysis of each attack, for calculating the attack complexity, it suffices to find a condition for δ which makes the attack successful.

We first describe and analyze the primal and dual attacks for the short secret variants of LWE, then transform the LWR instances into the LWE instances to apply the same attacks. These analyses show the relation between parameters and root Hermite factor δ.

Dual Attack. We are given $(A, \mathbf{b}) \in \mathbb{Z}_q^{m\times(n+1)}$ either from $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{D}_s)$, where the standard deviation of $\mathcal{D}_s$ is σ_s ($\mathcal{D}_s$ is either $\mathcal{HWT}_n(h_s)$ or $\mathcal{ZO}_n(\rho)$), or from $\mathcal{U}_q^{m\times(n+1)}$. In the original dual attack, an attacker constructs a lattice

$$\Lambda = \{(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}^m \times \mathbb{Z}^n : \mathbf{x}^T A = \mathbf{y}^T \pmod q\}$$

that is the orthogonal lattice of the matrix $(-A\|I_n)$ modulo q. One can find a short vector $\mathbf{v} = (\mathbf{x}, \mathbf{y})$ in Λ using BKZ and then check if $\langle \mathbf{x}, \mathbf{b}\rangle \pmod q$ is small or not. If $(A, \mathbf{b})$ is an $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{D}_s)$ instance with secret $\mathbf{s}$ and $\langle \mathbf{x}, \mathbf{b}\rangle$ is less than q in $\mathbb{Z}$, then $\langle \mathbf{x}, \mathbf{b}\rangle = \langle \mathbf{y}, \mathbf{s}\rangle + \langle \mathbf{x}, \mathbf{e}\rangle$ behaves as a Gaussian, otherwise it is distributed uniformly. Hence, if the attacker can find and collect short vectors $\mathbf{v} = (\mathbf{x}, \mathbf{y})$ in Λ such that $\langle \mathbf{x}, \mathbf{b}\rangle < q$, then the attacker would solve the distinguish problem.

Since the secret $\mathbf{s}$ is a (sparse) signed binary vector, the term $\langle \mathbf{y}, \mathbf{s}\rangle$ is somewhat smaller than $\langle \mathbf{x}, \mathbf{e}\rangle$. From this point, a tweaked strategy for this attack when the variances of the components in the secret vector $\mathbf{s}$ are significantly smaller than those of the error vector $\mathbf{e}$ arises as follows: We consider a weighted lattice

$$\Lambda' = \{(\mathbf{x}, \mathbf{y}') \in \mathbb{Z}^m \times (w^{-1} \cdot \mathbb{Z})^n : (\mathbf{x}, w \cdot \mathbf{y}) \in \Lambda\}$$

for some positive number $w > 0$. The optimal choice of w is

$$w = \frac{(\alpha q)}{\sqrt{2\pi}\sigma_s}$$

for reconciliation of variances $w^2 \cdot \sigma_s^2$ and $(\alpha q)^2/(2\pi)$ of $w \cdot s_i$ and e_j, respectively.

Let $\hat{q} = q/w = \sqrt{2\pi}\sigma_s \cdot \alpha^{-1}$. The lattice Λ' has the dimension $(m + n)$ and the volume $\hat{q}^n$. Hence, the BKZ algorithm outputs a short vector $\mathbf{v} = (\mathbf{x}, \mathbf{y}')$ of size $\|\mathbf{v}\| \approx \delta^{m+n} \cdot (\hat{q})^{\frac{n}{m+n}}$ which can be reduced down to $2^{2\sqrt{n \log \hat{q} \cdot \log \delta}}$ when $m+n = \sqrt{n \log \hat{q}/ \log \delta}$. Then $\langle \mathbf{x}, \mathbf{b}\rangle = \langle \mathbf{y}, w\cdot\mathbf{s}\rangle + \langle \mathbf{x}, \mathbf{e}\rangle$ is distributed as a Gaussian centered around zero and of standard deviation $\sigma = \|\mathbf{v}\| \cdot (\alpha q/\sqrt{2\pi})$ by central limit theorem (CLT). If $\sqrt{2\pi}\sigma < q$, then $\langle \mathbf{x}, \mathbf{b}\rangle$ can be distinguished from the uniform distribution modulo q with advantage about $\frac{1}{23}$ [6]. Therefore, the $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{D}_s)$ problem is secure only if

$$\frac{n \log \hat{q}}{\log^2 \alpha} \geq \frac{1}{4 \log \delta},$$

where $\hat{q} = \sqrt{2\pi}\sigma_s \cdot \alpha^{-1}$.

Example 1. In case that $\mathbf{s}$ is drawn from the distribution $\mathcal{HWT}_n(h_s)$, $\hat{q} = \sqrt{2\pi h_s/n} \cdot \alpha^{-1}$. If $\mathbf{s}$ is from the distribution $\mathcal{ZO}_n(\rho)$, then $\hat{q} = \sqrt{2\pi}\rho \cdot \alpha^{-1}$. Albrecht's combinatorial attack [2] for the small or sparse secret can be also applied in these cases so that we propose our parameters according to our attack combined with the combinatorial strategy.

Primal Attack. The key idea of the primal attack is the reduction from LWE to unique-SVP over a special lattice generated by a LWE instance. If the gap between λ_1 and λ_2 of this lattice is large enough, an attacker may find the shortest vector using the BKZ algorithm.

For a given $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{HWT}_n(h_s))$ instance $(A, \mathbf{b} = A\mathbf{s}+\mathbf{e}) \in \mathbb{Z}_q^{m\times(n+1)}$, construct the lattice

$$\Lambda = \{\mathbf{v} \in \mathbb{Z}^{n+m+1} : (A\|I_m\| - \mathbf{b})\,\mathbf{v} = 0 \pmod q\}$$

with the unique shortest vector $(\mathbf{s}, \mathbf{e}, 1)$. As with the case of dual attack, we consider the weighted lattice

$$\Lambda' = \{(\mathbf{x}, \mathbf{y}', z) \in \mathbb{Z}^n \times (w^{-1}\mathbb{Z})^m \times \mathbb{Z} : (\mathbf{x}, w \cdot \mathbf{y}, z) \in \Lambda\}.$$

for the constant $w = (\alpha q)/\sqrt{2\pi}\sigma_s$, which contains the unique shortest vector $(\mathbf{s}, w^{-1} \cdot \mathbf{e}, 1)$.

Let $\hat{q} = q/w = \sqrt{2\pi}\sigma_s \cdot \alpha^{-1}$. Since the lattice Λ' has the dimension $n + m + 1$ and the volume $\hat{q}^m$, we get $\lambda_2(\Lambda') \approx \sqrt{\frac{m+n+1}{2\pi e}}\ \hat{q}^{\frac{m}{m+n+1}}$ by the Gaussian heuristic. The attacker succeeds to find the unique-SVP solution $(\mathbf{s}, w^{-1}\mathbf{e}, 1)$ if

$$\frac{\lambda_2(\Lambda')}{\lambda_1(\Lambda')} \approx \frac{\sqrt{\frac{m+n}{2\pi e}}\hat{q}^{\frac{m}{m+n}}}{\sqrt{m+n}\cdot\alpha\hat{q}} = \frac{\alpha^{-1}}{\sqrt{2\pi e}\cdot\hat{q}^{\frac{n}{m+n}}} \geq \tau \cdot \delta^{m+n}$$

for a constant $0 < \tau < 1$. To minimize the complexity, an attacker may choose $m + n = \sqrt{\frac{n\log\hat{q}}{\log\delta}}$ which yields $\hat{q}^{\frac{n}{m+n}} \cdot \delta^{m+n} = 2^{2\sqrt{n\log\hat{q}\cdot\log\delta}}$. Therefore, the $\mathsf{LWE}_{n,m,q,\alpha}(\mathcal{D}_s)$ problem is secure against the primal attack only if

$$\frac{n\log\hat{q}}{\log^2\hat{\alpha}} \geq \frac{1}{4\log\delta},$$

where $\hat{\alpha} = (\sqrt{2\pi e}\cdot\tau)\alpha$ and $\hat{q} = \sqrt{2\pi}\sigma_s \cdot \alpha^{-1}$.

The constant τ is a constant that can be experimentally determined. For example, Gama and Nguyen [25] and Albrecht et al. [5] estimated τ within the range [0.18, 0.48] for some special lattices. Addressing the recent analysis in [1] for the primal attack, we concluded that the dual attack with BKW style combinatorial strategy is the best attack in our setting.

Dual and Primal attacks on LWR. Now we return to the LWR problem. Given an LWR instance $(A, \mathbf{b} = \lfloor(p/q)\cdot A\mathbf{r}\rceil) \in \mathbb{Z}_q^{m\times n} \times \mathbb{Z}_p^m$,

$$\frac{q}{p}\cdot\mathbf{b} = \frac{q}{p}\cdot\left\lfloor\frac{p}{q}\cdot A\mathbf{r}\right\rceil = A\mathbf{r} + \mathbf{t},$$

where $\mathbf{t} \in (-q/2p, q/2p]^m$. The rounding error $\mathbf{t}$ heuristically follows an uniform random distribution on $(-q/2p, q/2p]^m$. Therefore, in the view of attacker, the transformed instance $(A, (q/p)\cdot\mathbf{b})$ can be regarded as an LWE instance, and we apply the attacks on LWE to $(A, (q/p)\cdot\mathbf{b})$.

Since the variance of uniform random variable on $(-q/2p, q/2p]$ is $(q^2/12p^2)$, the parameter conditions to make LWR secure against the attacks can be obtained by substituting α with $p^{-1}\sqrt{\pi/6}$. The following inequalities are the conditions for $\mathsf{LWR}_{m,n+\ell,q,p}(\mathcal{HWT}_m(h_r))$ to be secure against the primal and dual attacks, respectively.

– Dual attack:

$$\frac{m \log \hat{q}}{\log^2 \hat{p}} \geq \frac{1}{4 \log \delta}$$

for $\hat{p} = \sqrt{6/\pi} \cdot p$ and $\hat{q} = p\sqrt{12 h_r / m}$.

– Primal attack:

$$\frac{m \log \hat{q}}{\log^2 \hat{p}} \geq \frac{1}{4 \log \delta}$$

for $\hat{q} = p\sqrt{12(h_r/m)}$ and $\hat{p} = (\sqrt{3/\pi^2 e} \cdot \tau^{-1})p$.

We concluded that the dual attack in [1] adjusted to our strategy is the best attack for LWR with sparse signed binary secret.

4.2.2 The BKZ Complexity

In this subsection, we explain how to set the root Hermite factor δ such that the attack complexities for given δ exceed 2^λ, where λ is the security parameter. We follow the strategies to measure the BKZ complexity in NewHope [7] and Frodo [13]. We review the relations among the root Hermite factor δ, the block size b, and the time complexity T for the BKZ algorithm in their paper as follows.

– (pessimistic) T can be estimated as 2^{cb} (about $b2^{cb}$ CPU cycles), where c is some constant. This is an approximate lower bound of the complexity for a single SVP calculation using the sieve algorithm [11, 29–31].
– $\delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)}$.

From this, if we fix the constant c, we can calculate δ from a given T. The best known constant is achieved by applying Grover's quantum search algorithm to the sieve algorithms [29, 31], which sets $c = 0.265$.

Hence, to make the attack using the BKZ algorithm as in Section 4.2.1 infeasible for security parameters $\lambda = 128$, $\lambda = 192$ and $\lambda = 256$, we should set the parameters such that the attack is successful only when $\delta \leq 1.00367$, $\delta \leq 1.00270$, and $\delta \leq 1.00216$, respectively.

4.2.3 Recommended Parameters

We chose parameter sets to achieve an infeasible attack complexity in following order: First, bound δ according to the time complexity T of desired security category, as seen in Section 4.2.2; Second, adjust parameters to make the best attack successful. We also chose parameters to achieve negligible decryption failure probability, in other words as mentioned on Lemma 1, each parameter set should achieve decryption failure probability less than $2^{-\lambda}$, where λ is the security parameter.

Note on Power-of-Twos. In particular, we set q and p as power-of-twos. In the LWE and LWR attacks, one can reduce the modulus q to $q' < q$ via modulus switching first and then apply arbitrary attack scenarios. Especially since we use the binary (and even sparse) secrets, the benefits in the considered attacks obtained by the modulus switching overwhelms others with strategies for specific q's as far as we know. Hence, any particular choice for modulus q does not harm the security. Therefore, we set q and p as power-of-twos to make the rounding procedures efficiently done through the bitwise shift process.

We chose 16 parameter sets: KEM_CATEGORYx_Ny for Lizard.KEM and CCA_CATEGORYx_Ny for Lizard.CCA, where (x,y) ∈ {(1,536), (1,663), (3,816), (3,952), (5,1088), (5,1300)}, and four sets

for both RLizard.KEM and RLizard.CCA called RING_CATEGORY1, RING_CATEGORY3_N1024, RING_CATEGORY3_N2048, and RING_CATEGORY5. We present the decryption failure probabilities[1] and attack complexities of LWE and LWR on our parameter sets in Table 1. The parameter sets are presented below Table 1.

Table 1: Decryption failure rate and attack complexities of each parameter set for the corresponding scheme: ϵ is the decryption failure probability and T_{LWE} and T_{LWR} are the time complexity of the best known attacks of LWE and LWR, respectively. The parameter sets RING_CATEGORY1, RING_CATEGORY3_N1024, RING_CATEGORY_N2048, RING_CATEGORY5 can be used for both RLizard.CCA and RLizard.KEM.

Parameter Set	$\log_2 \epsilon$	$\log_2 T_{\mathsf{LWE}}$	$\log_2 T_{\mathsf{LWR}}$
KEM_CATEGORY1_N536 CCA_CATEGORY1_N536	-159.212	133	130
KEM_CATEGORY1_N663 CCA_CATEGORY1_N663	-153.500	131	147
KEM_CATEGORY3_N816 CCA_CATEGORY3_N816	-304.467	193	195
KEM_CATEGORY3_N952 CCA_CATEGORY3_N952	-337.189	203	195
KEM_CATEGORY5_N1088 CCA_CATEGORY5_N1088	-381.331	266	257
KEM_CATEGORY5_N1300 CCA_CATEGORY5_N1300	-332.810	264	291
RING_CATEGORY1	-188.248	153	147
RING_CATEGORY3_N1024	-245.897	195	195
RING_CATEGORY3_N2048	-305.684	304	291
RING_CATEGORY5	-305.684	318	348

Parameter Sets of Lizard.CCA

CCA_CATEGORY1_N536
Security Classification : Category 1
$n = 536$
$m = 1024$
$q = 2048$
$p = 512$
$\ell = 256$
$d = 256$
$\texttt{\_16\_LOG\_Q} = 5$
$\rho = 1/2$
$h_r = 140$
$\texttt{CDF\_LENGTH} = 9$
$\texttt{CDF\_TABLE} = \{158, 148, 118, 81, 48, 22, 11, 4, 1\}$

[1] One can obtain the exact decryption failure rates, respectively, running a Python code reported at github: https://github.com/swanhong/LizardError.

CCA_CATEGORY1_N663
Security Classification : Category 1
$n = 663$
$m = 1024$
$q = 1024$
$p = 256$
$\ell = 256$
$d = 256$
_16_LOG_Q $= 6$
$\rho = 1/4$
$h_r = 128$
CDF_LENGTH $= 4$
CDF_TABLE $= \{918, 488, 74, 3\}$

CCA_CATEGORY3_N816
Security Classification : Category 3
$n = 816$
$m = 1024$
$q = 2048$
$p = 512$
$\ell = 384$
$d = 384$
_16_LOG_Q $= 5$
$\rho = 1/2$
$h_r = 200$
CDF_LENGTH $= 5$
CDF_TABLE $= \{304, 231, 100, 25, 4\}$

CCA_CATEGORY3_N952
Security Classification : Category 3
$n = 952$
$m = 1024$
$q = 2048$
$p = 512$
$\ell = 384$
$d = 384$
_16_LOG_Q $= 5$
$\rho = 1/4$
$h_r = 200$
CDF_LENGTH $= 6$
CDF_TABLE $= \{244, 204, 120, 49, 14, 3\}$

CCA_CATEGORY5_N1088
Security Classification : Category 5
$n = 1088$
$m = 2048$
$q = 4096$
$p = 1024$
$\ell = 512$
$d = 512$
_16_LOG_Q $= 4$
$\rho = 1/2$
$h_r = 200$
CDF_LENGTH $= 11$
CDF_TABLE $= \{264, 249, 214, 165, 115, 72, 41, 21, 10, 4\}$

CCA_CATEGORY5_N1300
Security Classification : Category 5
$n = 1300$
$m = 2048$
$q = 2048$
$p = 512$
$\ell = 512$
$d = 512$
_16_LOG_Q $= 5$
$\rho = 1/4$
$h_r = 200$
CDF_LENGTH $= 12$
CDF_TABLE $= \{526, 499, 427, 330, 230, 144, 82, 42, 19, 8, 3, 1\}$

Parameter Sets of Lizard.KEM

KEM_CATEGORY1_N536
Security Classification : Category 1
$n = 536$
$m = 1024$
$q = 2048$
$p = 512$
$\ell_1 = 16$
$\ell_2 = 16$
$d = 256$
_16_LOG_Q $= 5$
$\rho = 1/2$
$h_r = 140$
CDF_LENGTH $= 9$
CDF_TABLE $= \{158, 148, 118, 81, 48, 22, 11, 4, 1\}$

KEM_CATEGORY1_N663
Security Classification : Category 1
$n = 663$
$m = 1024$
$q = 1024$
$p = 256$
$\ell_1 = 16$
$\ell_2 = 16$
$d = 256$
$\texttt{\_16\_LOG\_Q} = 6$
$\rho = 1/4$
$h_r = 128$
$\texttt{CDF\_LENGTH} = 4$
$\texttt{CDF\_TABLE} = \{918, 488, 74, 3\}$

KEM_CATEGORY3_N816
Security Classification : Category 3
$n = 816$
$m = 1024$
$q = 2048$
$p = 512$
$\ell_1 = 24$
$\ell_2 = 16$
$d = 384$
$\texttt{\_16\_LOG\_Q} = 5$
$\rho = 1/2$
$h_r = 200$
$\texttt{CDF\_LENGTH} = 5$
$\texttt{CDF\_TABLE} = \{304, 231, 100, 25, 4\}$

KEM_CATEGORY3_N952
Security Classification : Category 3
$n = 952$
$m = 1024$
$q = 2048$
$p = 512$
$\ell_1 = 24$
$\ell_2 = 16$
$d = 384$
$\texttt{\_16\_LOG\_Q} = 5$
$\rho = 1/4$
$h_r = 200$
$\texttt{CDF\_LENGTH} = 6$
$\texttt{CDF\_TABLE} = \{244, 204, 120, 49, 14, 3\}$

KEM_CATEGORY5_N1088
Security Classification : Category 5
$n = 1088$
$m = 2048$
$q = 4096$
$p = 1024$
$\ell_1 = 32$
$\ell_2 = 16$
$d = 512$
$\texttt{\_16\_LOG\_Q} = 4$
$\rho = 1/2$
$h_r = 200$
$\texttt{CDF\_LENGTH} = 11$
$\texttt{CDF\_TABLE} = \{264, 249, 214, 165, 115, 72, 41, 21, 10, 4\}$

KEM_CATEGORY5_N1300
Security Classification : Category 5
$n = 1300$
$m = 1024$
$q = 2048$
$p = 512$
$\ell_1 = 32$
$\ell_2 = 16$
$d = 512$
$\texttt{\_16\_LOG\_Q} = 5$
$\rho = 1/4$
$h_r = 200$
$\texttt{CDF\_LENGTH} = 12$
$\texttt{CDF\_TABLE} = \{526, 499, 427, 330, 230, 144, 82, 42, 19, 8, 3, 1\}$

Parameter Sets of RLizard.CCA and RLizard.KEM

RING_CATEGORY1
Security Classification : Category 1
$n = 1024$
$q = 1024$
$p = 256$
$d = 256$
$\texttt{\_16\_LOG\_Q} = 6$
$h_s = 128$
$h_r = 128$
$\texttt{CDF\_LENGTH} = 4$
$\texttt{CDF\_TABLE} = \{382, 247, 67, 7\}$

RING_CATEGORY3_N1024
Security Classification : Category 3
$n = 1024$
$q = 2048$
$p = 512$
$d = 384$
$\texttt{\_16\_LOG\_Q} = 5$
$h_s = 256$
$h_r = 264$
$\texttt{CDF\_LENGTH} = 6$
$\texttt{CDF\_TABLE} = \{560, 443, 219, 68, 13\}$

RING_CATEGORY3_N2048
Security Classification : Category 3
$n = 2048$
$q = 2048$
$p = 512$
$d = 384$
$\texttt{\_16\_LOG\_Q} = 5$
$h_s = 184$
$h_r = 164$
$\texttt{CDF\_LENGTH} = 8$
$\texttt{CDF\_TABLE} = \{816, 720, 496, 266, 111, 36, 9, 2\}$

RING_CATEGORY5
Security Classification : Category 5
$n = 2048$
$q = 4096$
$p = 1024$
$d = 512$
$\texttt{\_16\_LOG\_Q} = 4$
$h_s = 256$
$h_r = 256$
$\texttt{CDF\_LENGTH} = 10$
$\texttt{CDF\_TABLE} = \{310, 289, 233, 162, 98, 51, 23, 9, 3, 1\}$

5 Implementation Aspects and Performance Figures

In this chapter, we describe and estimate performance and resource requirements of implementations on Intel x64 running Linux supporting the GCC compiler and on FPGA hardware.

5.1 Software Implementation

This section gives general implementation aspects and computational efficiencies of Lizard.KEM, RLizard.KEM, Lizard.CCA, and RLizard.CCA. We provide a reference and an optimized implementation in ANSI C, compiled with the GCC compiler.

For Lizard.KEM and RLizard.KEM, the files Lizard.c and RLizard.c are common for implementing the NIST API, including the functions `crypto_kem_keypair`, `crypto_kem_enc` and `crypto_kem_dec`. For Lizard.CCA and RLizard.CCA, the files Lizard.c and RLizard.c are common for implementing the NIST API, including the functions `crypto_encrypt_keypair`, `crypto_encrypt` and `crypto_encrypt_open`. To meet IND-CCA2 security, Lizard.KEM, RLizard.KEM, Lizard.CCA and RLizard.CCA should be used in public key pk as well as secret key sk in functions `crypto_kem_dec` and `crypto_encrypt_open`. Therefore, public key pk are included as part of the secret key in functions `crypto_kem_dec` and `crypto_encrypt_open` of file Lizard.c and RLizard.c.

To use various parameters in one implementation, we used the directing sentences `#if defined` and `#ifdef`. Therefore, we provide various parameters in file params.h. To use each parameter, you must use it by changing the annotation in the file params.h.

Additional files are provided file randombytes.c to use random values, file sha512.c to use the hash function and library, and file libkeccak.a of TupleHash256 to use variable length output.

5.1.1 Data Operations

Modulus Operation. For $x \in \mathbb{Z}_q$, rather than storing itself, we store the value (x <<_16_LOG_Q) where the data type of x is `uint16_t`, *i.e.*, the data is stored as the most significant $\log q$ bits in the 16-bit data space. In other words, we identify $\mathbb{Z}_q$ with the subspace of 16-bit data space of which the components are all zero except the most significant $\log q$ bits.

If vectors or matrices (*resp.* polynomials) are defined over $\mathbb{Z}_q$, then the above data storage strategy is applied to each of the components (*resp.* coefficient).

Rounding Operation. In this proposal, there are rounding operations $\lfloor (p/q) \cdot x \rceil$ over $\mathbb{Z}_p$ for some $x \in \mathbb{Z}_q$ and $\lfloor (2/p) \cdot y \rceil$ over $\mathbb{Z}_2$ for some $y \in \mathbb{Z}_p$. Note that x is stored as the most significant $\log q$ bits in the 16-bit data space, and the rounding output $\lfloor (p/q) \cdot x + 0.5 \rfloor \in \mathbb{Z}_p$ should be stored as the most significant $\log p$ bits in the same space. Therefore, the operation $\lfloor (p/q) \cdot x \rceil$ over $\mathbb{Z}_p$ is done by

$$(x + \texttt{RD\_ADD}) \ \wedge \ \texttt{RD\_AND}$$

where $\texttt{RD\_ADD} = 2^{15}/p$ and $\texttt{RD\_AND} = 2^{16} - 2^{16}/p$.

For example, when $p = 512$ and $q = 2048$, x can be represented as `0110 1000 1100 0000`, $\texttt{RD\_ADD} = 2^{15}/2^9 = 2^6$ will be `0000 0000 0100 0000` and $\texttt{RD\_AND} = 2^{16} - 2^{16}/2^9 = 2^6$ will be `1111 1111 1000 0000`. The operation $(x + \texttt{RD\_ADD}) \ \wedge \ \texttt{RD\_AND}$ will be as follows.

```
  0110 1000 1100 0000
+ 0000 0000 0100 0000
-------------------
  0110 1001 0000 0000

  0110 1001 0000 0000
∧ 1111 1111 1000 0000
-------------------
  0110 1001 0000 0000
```

The rounding operation $\lfloor (2/p) \cdot y \rceil$ over $\mathbb{Z}_2$ is done in exactly the same way.

5.1.2 Data Generations

Matrix and Vector Generations. As we generate matrices and vectors uniform randomly from the finite set $\mathbb{Z}_q^{m\times n}$, or following distributions, we introduce the algorithms for these random generations as follows.

First, we introduce how to generate the random matrix in $\mathbb{Z}_q^{m\times n}$ with the pseudorandom generator `randombytes()`. To achieve automatic reduction of a matrix modulo q, we set the data type of elements of a matrix as `uint16_t`, and left shift them `_16_LOG_Q` bits.

- $A \leftarrow \mathbb{Z}_q^{m\times n}$:
  - For $1 \le i \le m$ and $1 \le j \le n$, randomly generate the (i,j)-th component A_{ij} of A with `randombytes()` where the data type of A_{ij} is `uint16_t`
  - For $1 \le i \le m$ and $1 \le j \le n$, compute A_{ij} `<<_16_LOG_Q`
  - Output the matrix A

Next, we explain the algorithm of sampling the vector following the distribution $\mathcal{ZO}_n(1/2)$ (*resp.* $\mathcal{ZO}_n(1/4)$). The sampling is used for secret key generation of our Lizard.CCA and Lizard.KEM.

- $\mathbf{s} \leftarrow \mathcal{ZO}_n(1/2)$:
  - For $0 \le i \le n-1$, randomly generate two bits $x, y \in \{0,1\}$ with `randombytes()`
  - Set $s_i = 1$ if $x = 0$ and $y = 1$, $s_i = -1$ if $x = y = 0$, and $s_i = 0$ otherwise, where s_i is an i-th component of $\mathbf{s}$

- $\mathbf{s} \leftarrow \mathcal{ZO}_n(1/4)$:
  - For $0 \le i \le n-1$, randomly generate three bits $x, y, z \in \{0,1\}$ with `randombytes()`
  - Set $s_i = 1$ if $x = y = 0$ and $z = 1$, $s_i = -1$ if $x = y = z = 0$, and $s_i = 0$ otherwise, where s_i is an i-th component of $\mathbf{s}$

In our Lizard.CPA, we generate an ephemeral secret vector $\mathbf{r} \in B_{m,h_r}$ following the distribution $\mathcal{HWT}_m(h_r)$ in the encryption phase. When generating $\mathbf{r}$, we additionally generate the encoded values of $\mathbf{r}$; an array `r_idx` of h_r integers in $[0, m-1]$ which denote indices of non-zero components of $\mathbf{r}$, and an integer `neg_start` in $[0, h_r-1]$ which denotes a starting index of -1. If $i <$ `neg_start`, then the `r_idx[i]`-th component of $\mathbf{r}$ is 1, and if $i \ge$ `neg_start`, the `r_idx[i]`-th component of $\mathbf{r}$ is -1. We note that a vector $\mathbf{r}$ and a tuple of an array and an integer (`r_idx[`h_r`], neg_start`) match bijectively.

With this array encoding, we can evaluate the multiplication $A\mathbf{r} \in \mathbb{Z}_q^{n\times m}$ of a matrix $A \in \mathbb{Z}_q^{n\times m}$ and the vector $\mathbf{r}$ since `r_idx` contains the index information of non-zero components of $\mathbf{r}$. To be precise, we only read the `r_idx[i]`-th column of M for $0 \le i \le h_r$; add $\mathbf{a}_i$ if $i <$ `neg_start` and subtract $\mathbf{a}_i$ if $i \ge$ `neg_start`. That is, the number of `for` loops in the algorithm reduces from nm to nh_r.

- $\mathbf{r} \leftarrow \mathcal{HWT}_m(h_r)$
  - Set $hw = 0$ and $\mathbf{r}$ as a zero vector
  - Generate a random number $j \in [0, m-1]$ and a random bit `bit` $\in \{0,1\}$ with `randombytes()`
  - If $r_j = 0$, then set $r_j = 2 \cdot$ `bit` $- 1$ and hw += 1
  - Repeat the above algorithm until $hw < h_r$

- Generation of `r_idx`

- Set `neg_start` $= 0$ and `back_position` $= h_r$
- For $0 \leq i \leq m-1$, set `r_idx[neg_start]` $= i$ and `neg_start` $+= 1$ if $r_i = 1$, and `r_idx[back_position]` $= i$ and `back_position` $-= 1$ if $r_i = -1$
- Repeat the above algorithm until `neg_start` != `back_position`

In Lizard.KEM, RLizard.KEM, Lizard.CCA, and RLizard.CCA, the ephemeral secret vector $\mathbf{r} \in B_{m,h_r}$ (*resp.* matrix $R \in B^{\ell}_{m,h_r}$) should be deterministically generated by a Hash function with some input. Therefore, rather than using `randombytes()` whenever it is needed, we generate sufficiently long hash output at once and divide it to several blocks.

• $\mathbf{r} \leftarrow H($`input`$)$

- Get some input vector `input`, and compute the long hash value `Hash` = `TupleHash256`(`input`)
- Set $hw = 0$ and $\mathbf{r}$ as a zero vector
- Compute $j =$ `Hash` $\%\ m$ and left shift `Hash` for $\log m$ bits
- Compute `bit` = `Hash` $\%\ 2$ and left shift `Hash` for a bit
- If $r_j = 0$, then set $r_j = 2 \cdot$ `bit` $- 1$ and $hw\ += 1$
- Repeat the above algorithm until $hw < h_r$

The above algorithm $\mathbf{r} \leftarrow H($`input`$)$ is a case of Lizard.CCA and RLizard.CCA. For Lizard.KEM and RLizard.KEM, we sample the matrix $R \leftarrow H($`input`$)$ where each column vector of R is sampled from the above algorithm.

In the key generation phases of our schemes, we sample errors through the inversion sampling which uses a precomputed table for a discrete cumulative density function (CDF) over a small interval. We name process the `Sample_DG()` algorithm. The output distribution from this algorithm is a discrete bounded symmetric distribution which is very close to the discrete Gaussian distribution with respect to the Rényi divergence. More precisely, we preset a positive integer array `CDF_TABLE` of the length `TABLE_LENGTH` according to the CDF. Note that `CDF_TABLE` is an array of increasing positive integers, *i.e.*, `CDF_TABLE`$[i] \leq$ `CDF_TABLE`$[i+1]$ for $0 \leq i <$ `TABLE_LENGTH` $- 1$.

• `sample` $\leftarrow$ `Sample_DG()`

- Generate random numbers `rnd` $\in [0,$ `CDF_TABLE[TABLE_LENGTH - 1]`$]$ and `sign` $\in \{0,1\}$ with `randombytes()` where the data type of both numbers is `uint16_t`
- Find the smallest integer `sample` $\in [0,$ `TABLE_LENGTH` $- 1]$ such that `rnd` $\leq$ `CDF_TABLE`$[$`sample`$]$
- Compute `sample` $= ((-$`sign`$) \wedge$ `sample`$) +$ `sign`, *i.e.*, flip `sample` if `sign` $= 0$
- Output `sample`

Polynomial Generations. As a polynomial a corresponds to a vector $\mathbf{a} = (a_0, a_1, ..., a_{n-1})$ bijectively, we can match the polynomial ring R with the vector space $\mathbb{Z}^n$, and the quotient polynomial ring R_q with the vector space $\mathbb{Z}_q^n$. Therefore, we may regard the notation of a polynomial generation as a vector generation.

$$a = \sum_{i=0}^{n-1} a_i X^i \in R \Longleftrightarrow \mathbf{a} = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}^n$$

$$a = \sum_{i=0}^{n-1} a_i X^i \in R_q \Longleftrightarrow \mathbf{a} = (a_0, a_1, ..., a_{n-1}) \in \mathbb{Z}_q^n$$

From the vector-polynomial correspondence, we can regard polynomial generation as vector generation without specifying the bijection all the time. For example, we introduce the secret polynomial generation algorithm as follows:

• $s \leftarrow \mathcal{HWT}_n(h_s)$

- Set $hw = 0$ and $\mathbf{s}$ as a zero vector
- Generate a random number $j \in [0, n-1]$ and a random bit $\texttt{bit} \in \{0, 1\}$ with `randombytes()`
- If $s_j = 0$, then set $s_j = 2 \cdot \texttt{bit} - 1$ and $hw \mathrel{+}= 1$
- Repeat the above algorithm until $hw < h_s$
- Identify the vector $\mathbf{s}$ with the polynomial s thorough the vector-polynomial correspondence

We also generate encoded values of s, an array `s_idx` of h_s integers in $[0, n]$ and an integer `neg_start`.

• Generation of `s_idx`

- Set $\texttt{neg\_start} = 0$ and $\texttt{back\_position} = h_s$
- For $0 \le i \le m-1$, set $\texttt{s\_idx[neg\_start]} = i$ and $\texttt{neg\_start} \mathrel{+}= 1$ if $s_i = 1$, and $\texttt{s\_idx[back\_position]} = i$ and $\texttt{back\_position} \mathrel{-}= 1$ if $s_i = -1$
- Repeat the above algorithm until $\texttt{neg\_start} \mathrel{!}= \texttt{back\_position}$

5.1.3 Computational Efficiency

We report an optimized version of implementation tested under the following platform.

Linux: PC running Linux Ubuntu 14.04.3 LTS x86_64

CPU: Intel Xeon E5-2640 v3 at 2.60GHz, Octa core

Compiler: GCC 4.8.4 using `gcc -O3 -fomit-frame-pointer -msse2avx -mavx2 -march=native -std=c99`

For Lizard.KEM, RLizard.KEM, Lizard.CCA and RLizard.CCA, the parameter set supplies 128-bit, 192-bit and 256-bit security against all known quantum attacks. We present the parameter sets for various cases.

Operations	Parameter	SharedSecret (bytes)	Ciphertext (bytes)	Public Key (bytes)	Private Key (bytes)
Lizard.KEM	KEM_CATEGORY1_N536	32	17,696	1,130,496	8,608
	KEM_CATEGORY1_N663	32	10,896	1,390,592	10,640
	KEM_CATEGORY3_N816	48	26,928	1,720,320	19,632
	KEM_CATEGORY3_N952	48	31,280	1,998,848	22,896
	KEM_CATEGORY5_N1088	64	35,904	4,587,520	34,880
	KEM_CATEGORY5_N1300	64	42,688	2,727,936	41,664
RLizard.KEM	RING_CATEGORY1	32	2,080	4,096	385
	RING_CATEGORY3_N1024	48	4,144	4,096	641
	RING_CATEGORY3_N2048	48	8,240	8,192	625
	RING_CATEGORY5	64	8,256	8,192	769

Table 2: Size of Lizard.KEM and RLizard.KEM

Operations	Parameter	KeyGen (ms)	Enc (ms)	Dec (ms)
Lizard.KEM	KEM_CATEGORY1_N536	75.895	0.324	0.351
	KEM_CATEGORY1_N663	92.566	0.362	0.403
	KEM_CATEGORY3_N816	119.728	0.590	0.666
	KEM_CATEGORY3_N952	138.215	0.676	0.794
	KEM_CATEGORY5_N1088	306.368	0.846	0.905
	KEM_CATEGORY5_N1300	183.198	0.826	0.896
RLizard.KEM	RING_CATEGORY1	0.458	0.040	0.044
	RING_CATEGORY3_N1024	0.519	0.077	0.088
	RING_CATEGORY3_N2048	0.889	0.102	0.119
	RING_CATEGORY5	0.933	0.137	0.161

Table 3: Performance of Lizard.KEM and RLizard.KEM

Operations	Parameter	Plaintext (bytes)	Ciphertext (bytes)	Public Key (bytes)	Private Key (bytes)
Lizard.CCA	CCA_CATEGORY1_N536	32	1,648	1,622,016	137,216
	CCA_CATEGORY1_N663	32	983	1,882,112	169,728
	CCA_CATEGORY3_N816	48	2,496	2,457,600	313,344
	CCA_CATEGORY3_N952	48	2,768	2,736,128	365,568
	CCA_CATEGORY5_N1088	64	3,328	6,553,600	557,056
	CCA_CATEGORY5_N1300	64	3,752	3,710,976	665,600
RLizard.CCA	RING_CATEGORY1	32	2,208	4,096	257
	RING_CATEGORY3_N1024	48	4,272	4,096	513
	RING_CATEGORY3_N2048	48	8,496	8,192	369
	RING_CATEGORY5	64	8,512	8,192	513

Table 4: Size of Lizard.CCA and RLizard.CCA

Operations	Parameter	KeyGen (ms)	Enc (ms)	Dec (ms)
Lizard.CCA	CCA_CATEGORY1_N536	156.320	0.031	0.034
	CCA_CATEGORY1_N663	176.570	0.032	0.036
	CCA_CATEGORY3_N816	250.555	0.052	0.064
	CCA_CATEGORY3_N952	275.555	0.057	0.072
	CCA_CATEGORY5_N1088	663.879	0.062	0.086
	CCA_CATEGORY5_N1300	392.828	0.071	0.101
RLizard.CCA	RING_CATEGORY1	0.449	0.036	0.039
	RING_CATEGORY3_N1024	0.513	0.057	0.075
	RING_CATEGORY3_N2048	0.875	0.078	0.093
	RING_CATEGORY5	0.920	0.108	0.135

Table 5: Performance of Lizard.CCA and RLizard.CCA

The code uses only plain C instructions, without assembly nor SIMD instructions. For optimized speed, we used the loop fusion and loop unrolling methods. In optimized implementation, the code performs addition and subtraction operations to reduce the number of multiplication operations. For example, the optimized code performs the operation using `r_idx` instead of `r`.

On the platform above, we have presented the required space of Lizard.KEM and RLizard.KEM in Table 2 and the timing results in Table 3. We have also presented the required space of Lizard.CCA and RLizard.CCA in Table 4 and the timing results in Table 5. A certain amount of error is possible in Table 3 and Table 5 when implementing Lizard.KEM, RLizard.KEM, Lizard.CCA and RLizard.CCA.

5.2 Hardware Implementation

In this section, we propose the hardware architecture for Lizard Public Key Encryption and report the performance of the FPGA, which we implemented using Lizard.CPA and RLizard.CPA. These two Lizard modules mainly consist of a memory part and an addition part. Since the portion of the addition part is very small, while that of the memory part is very large, we decided to store only the data needed by calculation in the memory. Therefore, the operation of the module includes the data input/output process.

The advantage of Lizard PKE from the hardware implementation viewpoint is the simple calculation and ease of resource sharing. Since the q value is 2^{10}, setting the register Sum for storage as 10-bit only has the effect of becoming a modulus by itself. Since the key calculation is an accumulation that is a repetition of addition and subtraction, the calculation part is very simple, except for the storage space such as the memory. This means not only that the area is small but also that high-frequency operation is possible. The size of the area is even smaller than AES requires. Furthermore, it is easy to share resources since the various operational modes have similar hardware structures.

On the other hand, it requires a large storage space such as a cursor memory since the parameters are large, and the processes of inputting/outputting in a common size (32-bit word) and writing them to memory become complex because the volumes of data can differ considerably. One must also consider the fact that the use of memory is essential because of the large storage space.

Parameter of Lizard.CPA and RLizard.CPA For Lizard.CPA, the classical parameter set supplies 128-bit security against the classical attacks, but not enough against quantum attacks. The recommended parameter set provides 128-bit security against all known quantum attacks. The paranoid parameter set would remain secure and have 128-bit security against quantum attacks even if a remarkable improvement towards solving SVP arises. We present the parameter sets for the case that $\mathcal{D}_s = \mathcal{ZO}_n(1/2)$ and $\mathcal{D}_r = \mathcal{HWT}_m(128)$. We fix the plaintext modulus as $t = 2$ and $h_r = 128$. The following table is the suggested parameter sets for 128-bit security. For RLizard.CPA, we set $\mathcal{D}_s = \mathcal{D}_r = \mathcal{HWT}_n(128)$ and $\lambda = 128$.

Operations	m	n	$logq$	$logp$	α^{-1}
Lizard.CPA Classical	840	544	10	8	171
Lizard.CPA Recommended	940	608	10	8	182
Lizard.CPA Paranoid	1450	736	10	8	160
RLizard.CPA	1024	1024	10	8	154

Table 6: Parameter of Lizard.KEM and RLizard.KEM

We have implemented based on the recommended parameter of Lizard.CPA.

Architecture of Lizard.CPA The Fig. 1 shows the hardware architecture of Lizard.CPA.

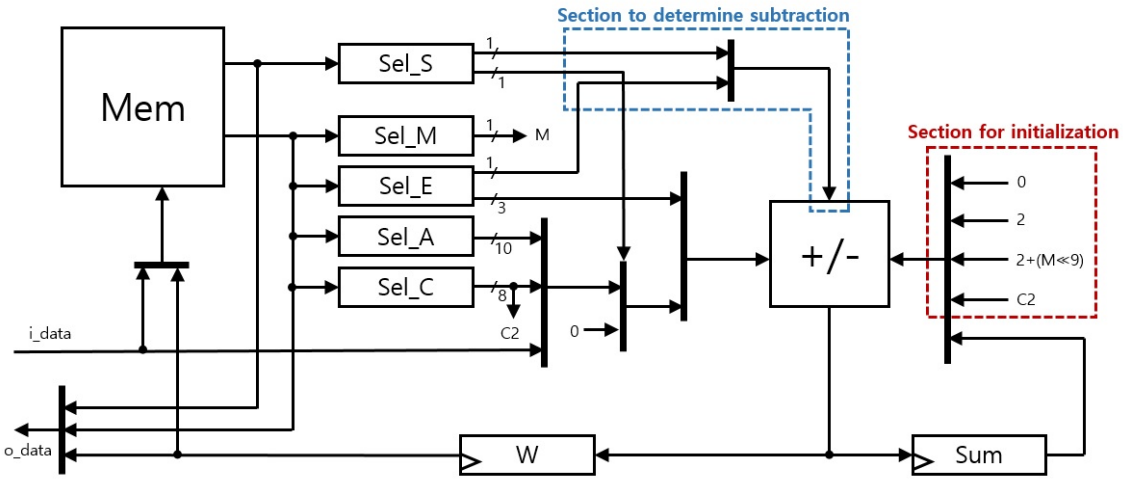


Fig. 1: Data path of Lizard.CPA

In the Fig. 1, Sel_S, Sel_M, Sel_E, Sel_A, and Sel_C are the multiplexers used to select the elements of S, $\mathbf{M}$, E, A and $\mathbf{c}$. The register Sum is the space for the accumulated data, while W is the storage space in which the final accumulated results are grouped into a 32-bit word. The adder is used to accumulate the value of the register Sum, the initial value of which is one of the inputs in the red box at the beginning of accumulation. The other input of the adder determines whether the output of Sel_S (S or $\mathbf{r}$) is added (S or $\mathbf{r}$ being 1), subtracted (S or $\mathbf{r}$ being –1), or does nothing (S or $\mathbf{r}$ being 0) to the result of the Sel_A (one of the elements of A for key generation), the result of Sel_C (one of elements of c_1 for decryption), or the outside input value (one of the elements of A or B for encryption). In the key generation process, there is an additional calculation involving either the addition or subtraction of the 3-bit value of the Sel_E output depending on the sign of the remaining 1 bit. The Fig. 1 omits the step in which two 9-bit data are converted into two 4-bit data containing the 3-bit data of 0 to 7 and the 1-bit data for the sign. It also omits the control circuit, which uses the registers and adders for the finite state machine and the counter.

Lizard.CPA requires three counters to count n, ℓ, and m, and the proposed design uses only one adder through resource sharing.

Finite State Machine of Lizard.CPA

Key Generation. The process begins with the inputting of all the values of S. The portions of 1, –1, and 0 of S are determined by the input from the outside (i.e. the same as for $\mathbf{r}$).

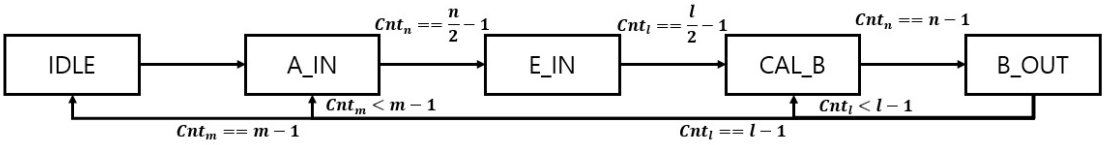


Fig. 2: Finite state machine for generating a key in Lizard.CPA

When the Lizard.CPA module is run in the key generation mode while S is being input, it receives A and E in words per row through A_IN and E_IN. The count in the module is incremented when a word is input, and the data are stored in the memory with the count value as the address. If S is input before the module starts, the address value of the word is specified at the same time for writing the data. However, since A and E use the internal counter as the address value, they must be input in sequence when they are input from the outside. If both a row for A and a row for E are inputted, a B element is calculated with S (which was input before the module started), the cycle n is ringed, and a row of B is calculated by repeating it ℓ times. Whenever two elements of B are calculated, they are grouped into a word and output to the outside. When a row of B is calculated, the next rows of A and E are input to calculate the next row of B. The key generation is completed when the process of inputting rows A and E and calculating row B is repeated m times, and m rows of B are all calculated.

Encryption. The process begins by inputting the R and M values in advance.

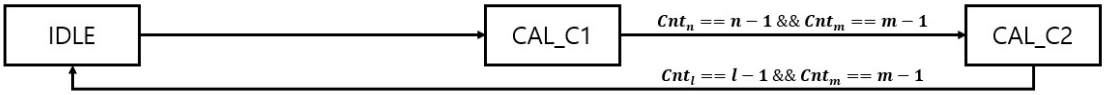


Fig. 3: Finite state machine for encryption of Lizard.CPA

When the module starts in the encryption mode, the module receives the elements of A in units of the row to calculate c_1, and only one element is input into each word as the elements are input in units of words. As such, A in the first of the n columns and m elements in the selected column are selected one at a time from the top. This contrasts with the fact that A, E, and B

are input or output in rows, and that two elements are transmitted into a word, during the key generation process. An element of c_1 is finally calculated by accumulating the calculation with $\mathbf{r}$, which was input before the module started, whenever A was input in the column, in the register Sum. The initial value of calculation of the c_1 element is 2, the accumulated value is maintained in 10 bits, and the final accumulated value is obtained from the top 8 bits by discarding the bottom 2 bits. Each time four elements of c_1 are calculated, they are grouped into a word and stored in the memory. In the same way, B is input in the row (l rows and m elements in each row) to calculate c_2. To calculate each element of c_2, the initial value is calculated by adding 2 to the 1 bit selected from each word of $\mathbf{M}$ (selected from the MSB) and shifted by 9 bits to the left. The process is finished with the calculation of c_2. The encryption process ends when the results c_1 and c_2 are stored in the memory. They can be read by specifying the register name and the address.

Decryption. The process begins by inputting S, c_1, and c_2 values in advance.

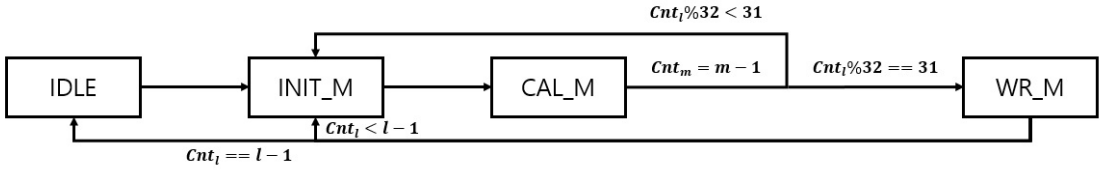


Fig. 4: Finite state machine for decryption of Lizard.CPA

The element of c_2 is initialized with the value of the register Sum by INIT_M, and CAL_M performs the accumulation using the S and c_1 values for n cycles. The top two bits of the final accumulated value are exclusive OR' ed to 1 bit of $\mathbf{M}$. While the process is repeated l times, 32 bits of $\mathbf{M}$ are stored in the WR_M step. Using the dual port memory means reading or writing the data of up to two data at a time. The limitation makes it necessary to use INIT_M and WR_M. If there is no limitation on data reading or writing by using the register instead of the memory, the steps of INIT_M and WR_M can be eliminated. As with the process of encryption, since the result is not output but saved in the memory during the process, $l/32 = 8$ words of $\mathbf{M}$ is read by specifying the register name and address after completing the process.

Architecture of RLizard.CPA The Fig. 5 shows the hardware architecture of RLizard.CPA.

As with Lizard.CPA, Sel_S, Sel_E, Sel_A and Sel_C in the Fig. 5 are the multiplexers used to select the elements of S, $\mathbf{M}$, E, A and $\mathbf{c}$ one at a time by selecting 2-bit, 1-bit, 4-bit, 10-bit and 8-bit. However, the method of Sel_M is different from that of Lizard.CPA since RLizard.CPA stores one byte of each word of A in the available space. In RLizard.CPA, only the difference of the coefficient with a value of -1 or 1 of S or $\mathbf{r}$ is stored in Mem1, and the coefficients of A, B, and c_1 are stored in Mem0. Let' s assume that the former is the difference data and the latter is the coefficient data. Since the difference data must be read first to determine the address of the coefficient data, it is necessary to read these two data sequentially. We used the dgr and neg registers to store the address of the coefficient data to read in the next clock cycle and the option of whether to subtract. This method is called pipe lining. It reads the difference data in the first cycle and then reads the coefficient data with the value of the dgr register as the address in the

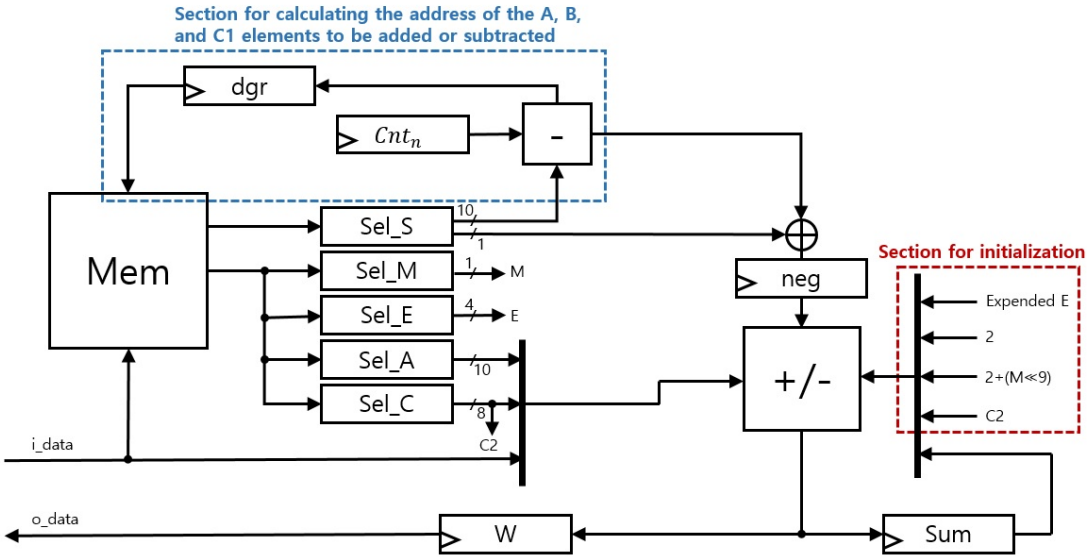


Fig. 5: Data path of RLizard.CPA

next cycle. It also reads the next difference data in the same cycle to be ready for the following clock cycle. It reads only the coefficient data in the last cycle. It differs from Lizard.CPA in that it can read E and use it as the initial value during the key generation cycle since it reads only the difference data in the first cycle. For simplicity, the Fig. 5 also omits the step in which 9-bit data input are converted into two 4-bit data of -7 to 7 and stored in the memory. Likewise, the register Sum stores the accumulation result; and W is the storage space in which each of the final accumulation results are grouped into a 32-bit word. The value in the register W is not written in the memory but rather is directly output. The Fig. 5 shows Cnt_n , which is one of the counters. The control circuit (not shown in the Fig. 5) has another counter for counting S and $\mathbf{r}$. Like the lattice-based Lizard, the proposed design uses only one adder through resource sharing.

Finite State Machine of RLizard.CPA Unlike Lizard.CPA, there is only one finite state machine in RLizard.CPA.

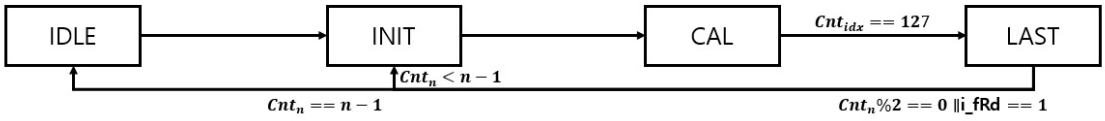


Fig. 6: Finite state machine in RLizard.CPA

All four steps perform multiplication of two n-degree polynomials through state transition. Although the process usually requires about n^2 cycles, the use of pipe lining requires only 129 clock

cycles. INIT initially reads only the difference data and the data needed for initialization (E for key generation, $\mathbf{M}$ for encryption 1, and c_2 for decryption). CAL reads the difference data and the coefficient data, while LAST reads only the coefficient data. The module performs addition or subtraction only in CAL and LAST, which read the coefficient data from the memory. The LAST block also groups two of the final results at a time and outputs them to the outside.

Performance

Latency. The following table shows the performance of the two Lizard modules.

Operations	Type	Computation	Performance (T(I) = 1, T(O) = 1)	Latency	
				@50MHz	@100MHz
Lizard.CPA	KeyGen	$m(n+\ell)T(I) + m\ell(n+T(O))$	150.5 M cycles	3 s	1.5 s
	Enc	$m(n+\ell)T(I)$	829.4 k cycles	16.6 ms	8.3 ms
	Dec	$\ell(n+1) + \ell/32$	155.9 k cycles	3.1 ms	1.6 ms
RLizard.CPA		$128(n + (1+T(O))/2)$	131.2 k cycles	2.6 ms	1.3 ms

Table 7: Latency of Lizard.CPA and RLizard.CPA

T(I) and T(O) represent the delay of input and output, respectively. The operating times of all four steps (Key Generation, Encryption0, Encryption1, Decryption) are the same in RLizard.CPA. However, when the Lizard.CPA module is run in Key Generation mode or Encryption mode, it receives E, A and B. Therefore it outputs a signal when it is time to get the input. If it reads the data in same clock cycle after it gets the signal, $T(O) = 1$. We performed the number of Cycles and Latency as $T(I) = 1$ and $T(O) = 1$.
The table above does not include the time required to input data in advance or read the data after the end of the process.

Area. The GE (Gate Equivalent) Table 8 is measured based on the implementation of the Samsung 65nm Library. It is the performance at 50MHz Frequency and much the same as the one at 100MHz Frequency. It is expected to have a similar area when operated on higher frequencies.

	Lizard.CPA			RLizard.CPA		
	Area		Size of Memory Space	Area		Size of Memory Space
	total	storage space		total	storage space	
memory	646.9 k	644.7 k	0x3000 words	99.7 k	98.3 k	1k words
register	3321.4 k	3319.5 k	0x2740 words	204.1 k	202.7 k	512 words + 64*22-bit

Table 8: Area of Lizard.CPA and RLizard.CPA

6 Advantages and limitations

In this section, we present our implementations of our scheme for special purposes. These results show that Lizard is flexible and efficient for various usage. The device we used in Section 6.1 was Samsung Galaxy S7. In Section 6.2 and Section 6.3, the implementations were written in

C, and performed on a Linux environment containing an Intel Xeon E5-2620 CPU running at 2.10GHz with Turbo Boost and Multithreading disabled. We used AVX2 vector instructions for optimizing the implementation of our schemes. The version of gcc compiler is 5.4.0, and we compiled our C reference implementation with flags `-O3 -fomit-frame-pointer -mavx2 -march=native -std=c99` for the `x86_64` architecture.

Through this section, the performances of key generation (*resp.* encryption and decryption) of our schemes were reported as a mean value across 100 (*resp.* 100000) measurements. We recorded public key sizes of our schemes used in our software[2].

6.1 Application on Smartphone

Since the smartphone is one of the most commonly used devices, it is natural to consider a mobile implementation. We have implemented Lizard.CPA as an Android application. The parameters of the implementation satisfy 128-bit quantum security with bigger decryption failure probability. The performance of the application was comparable to computer implementation. The application used a small amount of memory (less than 20 megabytes), and used only one core of CPU. Therefore, we can see that Lizard is suitable for a smartphone.

Table 9: Parameter of Lizard.CPA on Android application implementation

m	n	$\log q$	$\log p$	α^{-1}	ρ	h_r
960	608	10	8	182	1/2	128

Table 10: Performance of Lizard.CPA on Android application implementation

KeyGen (ms)	Enc (ms)	Dec (ms)
288.618	0.0770	0.0229

6.2 Suitability for Small Message Space

Lizard can be utilized on low-end devices. We implemented our Lizard.CPA scheme with 32-bit message space under 128-bit classical security (119-bit quantum security). We used classical parameters suggested in Table 11, and set $\ell = 32$ to specify the message space. In general case, the public key size is 741kB, and an encryption takes only 0.009 milliseconds. The public key size can be reduced to 46kB if we replace the public matrix A by a 256-bit seed that generates A, and an encryption gets slower to 0.052 milliseconds while a decryption takes the same time. These performance data show us that Lizard can be efficient on low capacity devices.

[2] We can generate matrix A in our public key from a 256-bit seed with Pseudo-Random Generator (PRG) and store only the seed. To implement this case, we use `AES128` in the ECB mode in our implementation to expand a 256-bit seed, enabling the AES-NI instruction.

Table 11: Parameter of Lizard.CPA with 32-bit message space with 128-bit classical security

m	n	$\log q$	$\log p$	α^{-1}	ρ	h_r	ϵ
724	480	11	9	303	1/2	128	2^{-154}

Table 12: Performance of Lizard.CPA with 32-bit message space with 128-bit classical security

	ctxt (bytes)	pk (bytes)	sk (bytes)	KeyGen (ms)	Enc (ms)	Dec (ms)
A as matrix	576	741,376	3,840	4.749	0.009	0.001
(A as seed)		(46,368)		(1.891)	(0.052)	

6.3 Additive Homomorphic Encryption

Lizard can also be used as a post-quantum alternative for additive homomorphic encryption (AHE) which support the bounded number of homomorphic additions. Lizard.CPA can be naturally seen as an additive homomorphic encryption supporting the bounded number of additions together with the following addition procedure:

– $\mathsf{Lizard.CPA.Add}(\mathbf{c}_1, \cdots, \mathbf{c}_k)$: Output $\sum_{i=1}^{k} \mathbf{c}_i \in \mathbb{Z}_p^{n+\ell}$

Corollary 1 (Correctness). *The additive homomorphic encryption described above works correctly for k homomorphic additions as long as the following inequality holds for security parameter λ:*

$$Pr\left[|\langle \mathbf{e}, \mathbf{r}\rangle + \langle \mathbf{s}, \mathbf{f}\rangle| \geq \frac{q}{2tk} - \frac{q}{2pk}\right] < \mathsf{negl}(\lambda)$$

where $\mathbf{e} \leftarrow \mathcal{D}G_\sigma^m$, $\mathbf{r} \leftarrow \mathcal{H}WT_m(h_r)$, $\mathbf{s} \leftarrow \mathcal{Z}O_n(\rho)$, and $\mathbf{f} \leftarrow \mathbb{Z}_{q/p}^n$.

Proof. This is easily proved by Lemma 1 and the triangle inequality.

Parameters for Additive Homomorphic Encryption. It is harder to meet the correctness condition in Corollary 1 than the plain Lizard scheme. We suggest a parameter set for 128-bit quantum security that allows 100 additions as Table 13.

Table 13: Parameter for additive homomorphic encryption

m	n	$\log q$	$\log p$	α^{-1}	ρ	h_r
1024	816	16	14	21000	1/2	136

For this parameter set, the decryption failure probability after 100 homomorphic additions is approximately 2^{-29}.

Previously proposed additive homomorphic encryption schemes [27, 35, 36] of which performances are summarized in [19][3] can afford much more homomorphic additions with fixed param-

[3] In [19], they also suggested an AHE scheme with excellent performances, but their parameters are turned out to be insecure [23].

eter sets than ours. However, when one needs only bounded number of homomorphic additions, Lizard might provide a better trade-off so that it can be faster than other AHE schemes. For Lizard which supports 100 homomorphic additions, an encryption, decryption, and homomorphic addition take only 0.014, 0.012, and 0.0005 milliseconds, which are at least 147, 333, and 4 times faster than all of those of AHE schemes in [27, 35, 36], respectively. We present a sample result for 256-bit messages and 128-bit quantum security in Table 14.

Table 14: Performance of Lizard with 256-bit messages which supports 100 homomorphic additions

	ctxt (bytes)	pk (bytes)	sk (bytes)	KeyGen (ms)	Enc (ms)	Dec (ms)	Add (ms)
A as matrix	1,876	2,195,456	52,224	25.923	0.014	0.012	0.0005
(A as seed)		(524,320)		(21.444)	(0.092)		

References

1. Martin Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the Expected Cost of Solving uSVP and Applications to LWE. 2017. to appear.
2. Martin R Albrecht. On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. *IACR Cryptology ePrint Archive*, 2017:047, 2017.
3. Martin R Albrecht, Carlos Cid, Jean-Charles Faugere, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Designs, Codes and Cryptography*, 74(2):325–354, 2015.
4. Martin R Albrecht, Jean-Charles Faugere, Robert Fitzpatrick, and Ludovic Perret. Lazy modulus switching for the BKW algorithm on LWE. In *International Workshop on Public Key Cryptography*, pages 429–445. Springer, 2014.
5. Martin R Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In *International Conference on Information Security and Cryptology*, pages 293–310. Springer, 2013.
6. Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
7. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange—A New Hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, August 2016. USENIX Association.
8. Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited. In *Advances in Cryptology–CRYPTO 2013*, pages 57–74. Springer, 2013.
9. Shi Bai and Steven D Galbraith. Lattice decoding attacks on binary LWE. In *Australasian Conference on Information Security and Privacy*, pages 322–337. Springer, 2014.
10. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.
11. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 10–24. SIAM, 2016.
12. Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In *Theory of Cryptography Conference*, pages 209–224. Springer, 2016.
13. Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1006–1018, New York, NY, USA, 2016. ACM.

14. Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In *Public-Key Cryptography–PKC 2013*, pages 1–13. Springer, 2013.
15. Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 575–584. ACM, 2013.
16. Yuanmi Chen and Phong Q Nguyen. BKZ 2.0: Better lattice security estimates. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.
17. Jung Hee Cheon, Kyoohyung Han, Jinsu Kim, Changmin Lee, and Yongha Son. Practical post-quantum public key cryptosystem based on LWE. In *the 19th Annual international Conference on Information Security and Cryptology*, 2016. Available at `https://eprint.iacr.org`.
18. Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yong Soo Song. Lizard: Cut off the tail!//practical post-quantum public-key encryption from lwe and lwr. *IACR Cryptology ePrint Archive*, 2016:1126, 2016.
19. Jung Hee Cheon, Hyung Tae Lee, and Jae Hong Seo. A new additive homomorphic encryption based on the Co-ACD problem. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 287–298. ACM, 2014.
20. Alexander W Dent. A designer's guide to kems. *Lecture notes in computer science*, pages 133–151, 2003.
21. Alexandre Duc, Florian Tramèr, and Serge Vaudenay. Better algorithms for LWE and LWR. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 173–202. Springer, 2015.
22. Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015*, pages 617–640. Springer, 2015.
23. Pierre-Alain Fouque, Moon Sung Lee, Tancrède Lepoint, and Mehdi Tibouchi. Cryptanalysis of the Co-ACD assumption. In *Annual Cryptology Conference*, pages 561–580. Springer, 2015.
24. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Crypto*, volume 99, pages 537–554. Springer, 1999.
25. Nicolas Gama and Phong Q Nguyen. Predicting lattice reduction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 31–51. Springer, 2008.
26. 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`.
27. Marc Joye and Benoit Libert. Efficient cryptosystems from $2k$-th power residue symbols. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 76–92. Springer, 2013.
28. Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for LWE with applications to cryptography and lattices. In *Annual Cryptology Conference*, pages 43–62. Springer, 2015.
29. Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, PhD thesis, Eindhoven University of Technology, 2015. `http://www.thijs.com/docs/phd-final.pdf`, 2015.
30. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *Annual Cryptology Conference*, pages 3–22. Springer, 2015.
31. Thijs Laarhoven, Michele Mosca, and Joop Van De Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77(2-3):375–400, 2015.
32. Mingjie Liu, Xiaoyun Wang, Guangwu Xu, and Xuexin Zheng. Shortest lattice vectors in the presence of gaps. *IACR Cryptology ePrint Archive*, 2011:139, 2011.
33. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.
34. Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.
35. Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

36. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.
37. Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2009.
38. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Annual International Cryptology Conference*, pages 554–571. Springer, 2008.
39. Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 187–196. ACM, 2008.
40. 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.
41. Markus Rückert and Michael Schneider. Estimating the security of lattice-based cryptosystems. Cryptology ePrint Archive, Report 2010/137, 2010. `http://eprint.iacr.org/2010/137`.
42. Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1-3):181–199, 1994.
43. Ehsan Ebrahimi Targhi and Dominique Unruh. Quantum Security of the Fujisaki-Okamoto and OAEP Transforms. Cryptology ePrint Archive, Report 2015/1210, 2015. `http://eprint.iacr.org/2015/1210`.