Copyright ©2018 The Institute of Electronics,
Information and Communication Engineers

SCIS 2018 2018 Symposium on
Cryptography and Information Security
Niigata, Japan, Jan. 23 - 26, 2018
The Institute of Electronics,
Information and Communication Engineers

Prey on Lizard: Mining Secret Key on Lattice-based Cryptosystem

Seongho Han* Nakjun Choi* Hyeongcheol An*
Rakyong Choi† Kwangjo Kim*†

Abstract: With the development of quantum computers, post-quantum cryptography has been researched in the last decades. Lattice-based cryptography is one of the most fascinating candidates of post-quantum cryptography. This is due to the average and worst case provable security on lattice such as Learning with Errors(LWE) and Learning with Rounding(LWR). Lattice-based encryption scheme called Lizard based on LWE and LWR by Cheon *et al.* was suggested as a candidate public key cryptosystem for long-term security according to call-for-post quantum cryptography by NIST recently. Lizard was suggested to have great performance and high level of security. However, Lizard could be exploited because of its C implementation. In this paper, we investigate the way to break Lizard by side channel attacks such as timing and fault attacks. From these attacks, we can find secret key from source code. Finally, we propose countermeasures to protect Lizard from our attacks.

Keywords: Lattice-based Cryptography, Learning with Errors, Learning with Rounding, Timing Attack, Fault Attack

1 Introduction

1.1 Motivation

Public key cryptography provides authentication, integrity, and non-repudiation for secure services. RSA cryptosystem is one of the most well-known cryptosystems. RSA relies its security on the difficulty of Integer Factorization Problem(IFP). According to the current computational power of classical computers, it is believed to be impossible to decompose the 2048-bit key into two prime factors within a feasible time. However, when a quantum computer is developed, security cannot be guaranteed by Shor's algorithm [1] which solves IFP in polynomial time.

In response to the developments of quantum computer, NIST proposed a project to develop a new quantum-resistant cryptosystem [2]. Lattice-based cryptography is one of the most prominent candidates for post-quantum cryptography. Lattice-based cryptography is based on the hardness of Small Integer Solution(SIS) or Learning with Errors(LWE) problems. In 2005, Regev [3] suggested LWE problem and its reduction to worst-case hardness problem. LWE problem can be applicable to many lattice-based cryptographic protocols. For example, NTRUEncrypt is accepted as the IEEE standard [4]. Because of the speed of NTRUEncrypt cryptosystem and low memory-use, it can be applied to mobile devices and smart cards. In addition to the NTRU system, many systems such as BLISS [5] and BCNS [6] have been proposed.

Although LWE-based public key encryption has many advantages, the encryption phase is relatively slow compared to the decryption process since large parameter sizes are required for leftover hash lemmas or expensive Gaussian sampling. Banerjee and Peikert [7] proposed Learning with Rounding(LWR) in 2011 to solve this issue. Unlike LWE, LWR is deterministic and allows faster encryption.

Lizard proposed by Cheon *et al.*[8] combines both LWE and LWR problems. Lizard uses faster and fewer ciphertexts than Regev's scheme [9] or Lindner-Peikert's scheme [10] proposed in 2011. In terms of security, Lizard guarantees IND-CPA under appropriate parameters. However, some vulnerabilities were found in the implementation process of Lizard.

In this paper, we exploit the vulnerability of Lizard. We recovered the key in real time through side channel attacks such as timing and fault attacks. We found a correlation between encryption time and the length of the message from timing attack. Secret key was actually recovered by fault attack. Fault attacks are usually performed on hardware, but in this paper, we carried software fault attacks including skipping, zeroing, and randomizing which are proposed by Bindel [11].

We propose a countermeasure to prevent attacks. As a countermeasure against the timing attack, we should make the size of all ciphertexts equal. We successfully prevent timing attack from implementing this countermeasure. To prevent fault attack, we need to insert error-checking procedure in the implementation.

* Graduate School of Information Security, KAIST. 291, Daehak-ro, Yuseong-gu, Daejeon, South Korea 34141. {$hansh$09, cnj8160, anh1026, kkj}@kaist.ac.kr)
† School of Computing, KAIST. 291, Daehak-ro, Yuseong-gu, Daejeon, South Korea 34141. {$thepride$, kkj}@kaist.ac.kr

1.2 Outline of the Paper

In Section 2, we describe LWE and LWR, and then explain Lizard in brief. Section 3 introduces previous work about side channel attacks on lattice-based cryptography. We describe experimental setup in Section 4. In Section 5, we describe timing attack and fault attack. We propose countermeasures for our attacks in Section 6, discuss further remarks in Section 7, and make a conclusion in Section 8.

2 Background

2.1 Learning with Errors (LWE)

LWE is a lattice-based cryptographic problem against quantum computers. Since Regev [3] proposed LWE, a number of cryptographic protocols have been introduced. LWE problem is categorized into search-LWE and decisional-LWE.

LWE distribution is defined as follows: For a secret vector $s \in \mathbb{Z}_q^n$, LWE distribution $A_{s,\chi}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is sampled by choosing $a \in \mathbb{Z}_q^n$ uniformly at random and $e \leftarrow \chi$, and outputting:

$$(a, b = \langle s, a \rangle + e \ mod \ q) \tag{1}$$

Search-LWE: given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ drawn from $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$, find s.

Decisional-LWE: given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where every sample is distributed from either:
(1) $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$
(2) The uniform distribution
Distinguish whether samples are from (1) LWE distribution or (2) uniform distribution (with non-negligible advantage)

In many cases, decisional-LWE problem is used as the basis for a cryptosystem. The difficulty of the decisional-LWE problem is ensured by the worst case hardness of the standard lattice problem: the decisional version of the shortest vector problem (GapSVP), and the shortest independent vectors problem (SIVP).

2.2 Learning with Rounding (LWR)

LWR problem [7] is a derandomized version of LWE problem. That is, the error term is chosen deterministically. LWR problem is also categorized into search-LWR and decisional-LWR like LWE problem.

LWR distribution is defined as follows: For a secret vector $s \in \mathbb{Z}_q^n$, LWR distribution $A_{s,\chi}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_p$ is sampled by choosing $a \in \mathbb{Z}_q^n$ uniformly at random and $e \leftarrow \chi$, and outputting

$$(a, \lfloor \frac{p}{q} \cdot (\langle s, a \rangle \ mod \ q) \rceil) \in \mathbb{Z}_q^n \times \mathbb{Z}_p \tag{2}$$

Search-LWR is defined as follows: given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_p$ drawn from $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$, find s.

On the other hand, decisional-LWR is defined as follows: given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_p$ where every sample is distributed according to either:
(1) $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$
(2) The uniform distribution
Distinguish whether samples are from (1) LWR distribution or (2) uniform distribution (with non-negligible advantage)

For the appropriate variables, decisional-LWR is at least as difficult as decisional-LWE.

2.3 Lizard

Lizard [8] is a public key encryption scheme that combines LWE and LWR by removing several least significant bits of each computed vector. The public key consists of m instances of n-dimensional LWE samples, and $(n+1)$ instances of m-dimensional LWR samples where l is the dimension of plaintext vectors. The scheme is designed as follows:

Setup

1) Choose positive integers m, n, q, p, t and l.

2) Choose long-term secret key distribution $\mathcal{D}_s$ over $\mathbb{Z}^n$, ephemeral secret key distribution $\mathcal{D}_r$ over $\mathbb{Z}^m$, and parameter σ for discrete Gaussian distribution χ_σ.

3) Output $params \longleftarrow (m, n, q, p, t, l, \mathcal{D}_s, \mathcal{D}_r, \sigma)$

Key Generation

1) Generate a random matrix $A \leftarrow \mathbb{Z}_q^{m \times n}$. Choose a secret matrix $S = (s_1 \parallel \cdots \parallel s_l)$ by sampling column vectors $s_i \in \mathbb{Z}^n$ independently from the distribution $\mathcal{D}_s$.

2) Generate an error matrix $E = (e_1 \parallel \cdots \parallel e_l)$ from $\chi_\sigma^{m \times l}$. Let $B \leftarrow AS + E \in \mathbb{Z}_q^{m \times l}$ where the operations are held in modular q.

3) Output the public key $pk \leftarrow (A \parallel B) \in \mathbb{Z}_q^{m \times (n+l)}$ and secret key $sk \leftarrow S \in \mathbb{Z}^{n \times l}$

Encryption

1) For a plaintext $m = (m_i)_{1 \le i \le l} \in \mathbb{Z}_t^l$, choose an m-dimensional vector $r \in \mathbb{Z}^m$ from the distribution $\mathcal{D}_r$.

2) Compute the vectors $c_1' \leftarrow A^T r$ and $c_2' \leftarrow B^T r$ over $\mathbb{Z}_q$.

3) Output the vector $c \leftarrow (c_1, c_2) \in \mathbb{Z}_p^{n+l}$, where $c_1 \leftarrow \lfloor (\frac{p}{q}) \cdot c_1' \rceil \in \mathbb{Z}_p^n$, $c_2 \leftarrow \lfloor (\frac{p}{t} \cdot m + (\frac{p}{q}) \cdot c_2' \rceil \in \mathbb{Z}_p^l$

Decryption

1) For a ciphertext $c = (c_1, c_2) \in \mathbb{Z}_p^{n+1}$, compute and output the vector $m' \leftarrow \lfloor \frac{t}{p}(c_2 - S^T c_1) \rceil$.

2) Check whether $m = m'$

This scheme is IND-CPA secure under the hardness assumption of $LWE_{n,m,q,\chi_\sigma}(\mathcal{D}_s)$ and $LWR_{m,n+1,q,p}(\mathcal{D}_r)$.

Lizard ensures 128-bit security level. Security level is measured by BKZ algorithm used for Newhope [12] and Frodo [13].

3 Previous Work

In general, breaking cryptographic algorithms by software attack is considered to be a great success but difficult to achieve. But side channel attacks are beneficial to find various weaknesses. Lizard is a relatively recent scheme introduced in 2016. To the best of our knowledge, the attack for Lizard has not been published yet. So we need to look for the possible attacks for quantum-resistant schemes like Lizard.

In this section, we introduce several side channel attacks on post quantum algorithms such as NTRU.

3.1 Timing Attacks on NTRUEncrypt

Hoffstein *et al.*[14] proposed a ring-based public key cryptosystem (NTRU) for the first time in 1996. NTRU is a post quantum cryptography consisting of two algorithms: NTRUEncrypt for encryption and NTRUSign for digital signatures. Because it is based on lattice, a lot of previous researchs [15, 16] focused on attacking lattice itself has been done. In addition to this, a side channel attack against NTRU has also been attempted.

Silverman *et al.*[17] introduced a timing attack on NTRUEncrypt based on variation in the number of hash calls made on decryption. As part of the attack, the attacker performs some amount of precomputation, then submit a small number of constructed ciphertexts for decryption and measures the decryption times. After that, the attacker compares the precomputed result with the measured time. If there are few hash calls in the decryption process, it will take less time, otherwise it will take more time. Using these results can greatly reduce the time required to recover the key. The authors find that an attacker could recover a single key with about $\frac{k}{2}$-bit of effort. Additionally, they proposed a simple way to prevent these attacks by ensuring that all operations take a constant number of SHA calls.

3.2 Fault Attacks on NTRUEncrypt

Kamal *et al.*[18] introduced a fault analysis on NTRUEncrypt cryptosystem in 2011. In 2012, they [19] also proposed a fault analysis on NTRUSign digital signature scheme. In NTRUEncrypt, they assumed that the attacker can inject fault to a small number of coefficients of the polynomial input during decryption process. Figure 1 describes this decryption process in [18]. After injecting a small number of coefficients, the attacker can calculate the output of the faulty decryption process. Then the right secret key is determined by performing the encryption process on the ciphertext. Finally, compare it with the original message. They proposed two approaches to prevent these fault attacks. The first is to use algorithm level redundancy. And the second is to add parity bits on decryption process.

3.3 Power Analysis Attacks on NTRUEncrypt

Wang *et al.*[20] attempted power analysis attacks on NTRU-based wireless networks. Figure 2 illustrates the result of the power analysis attack in [20]. The first and second graphs shows the two average trace

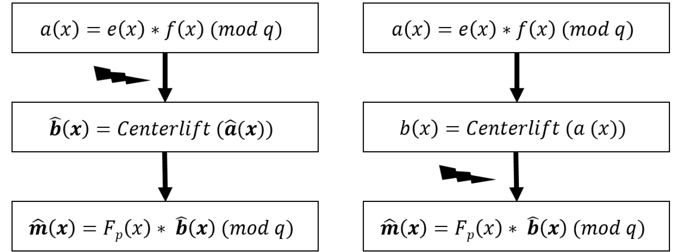

Figure 1: The decryption process after inducing faults: a) before the centerlift operation or b) after the centerlift operation.

and the last graph shows their differential trace. Because there are clear differences, the attacker can easily recover the secret key. To defend against these threats, they presented three countermeasures; random delays, masking, and dummy operations.

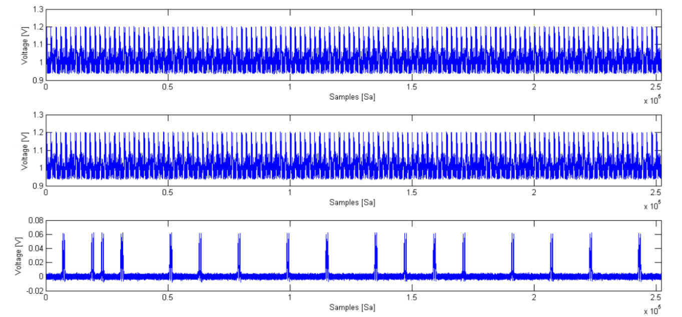

Figure 2: Differential Power Analysis on NTRU

Song *et al.*[21] also presented a power analysis attacks on NTRUEncrypt in 2009 and Lee *et al.*[22] described its countermeasures in detail in 2010. Atici [23] proposed the same attacks for RFIDs.

3.4 Other Side Channel Attacks

Kamal *et al.*[19] applied the scan-based side channel attack to NTRUEncrypt in 2012. They focused on scan-based Design-for-Test (DFT). Using this technique, they can obtain the secret information from cryptographic hardware devices. Recently Paterson [24] proposed a cold-boot attack on NTRU. Cold Boot Attack [25] is an attack that exploits the fact that if an attacker freezes dynamic RAM, the data is held for a while. Paterson investigated NTRU to see if the secret key was stored in memory, and developed an algorithm that effectively recovered the key. Like these systems, post quantum algorithms can be easily attacked by some side channel attacks. Since Lizard scheme has been released recently, such attacks have not been considered in the implementation of Lizard yet. But like NTRU, we can guess that Lizard can be vulnerable to such side channel attacks.

4 Experimental Setup

In this section we describe our experimental environment in detail and specify the values of the parameters that we have changed arbitrarily by injecting faults. And we define an attack model for our attacks.

4.1 Environment

Our experimental environment is as follows: Intel(R) Xeon(R) CPU E3-1220 v3 @ 3.10GHz, RAM 16.0GB, Ubuntu Linux 64-bit v16.04.3. The compiler is visual studio code and uses gcc v5.4.0. We used the source code of Lizard scheme in Github[1]. There are five sets of parameters: *Classical*, *Recommended*, *Homadd*, *ClassicalPlainText32bit*, and *CCA*. We chose the *Recommended* parameter set, where dimensions for LWE, LWR, and messages are 536, 1024, and 256, respectively. We performed a timing attack with these parameters. However, in case of fault attack, we transformed this setting into 10, 10, and 20 dimensions for smooth inverse operation.

4.2 Attack Model

Our attack model is assumed to be whitebox model. So we assume that the attacker knows everything except some secret elements. Due to the characteristic of LWE and LWR-based Lizard schemes, the secret matrix S and the error matrix E are not disclosed in the Key Generation part. Also, when the message is m-bit in the encryption part, m-dimensional vector r is not disclosed. However, all other parameter values are public and the attacker has full knowledge of the source code. This assumption allows a very strong power to the attacker.

5 Attack

In this section, we perform timing and fault attacks on Lizard. Since there was a time difference according to the amount of computation, the timing attack could be successfully performed. In addition, our attack model has no difficulty in performing a fault attack because we can freely access the source code. So we attempted three types of fault attacks and described the results for each.

5.1 Timing Analysis

We first checked the existence of time differences that could be distinguished in Lizard. In general, timing attacks are very useful when there are control statements like if() statements in the algorithm. The amount of computation of the scheme varies greatly depending on whether or not these control conditions are executed. As a result, the difference between the encryption and decryption time of the original message or the ciphertext become clear. By analyzing these parts, an attacker can easily obtain a secret key. But unfortunately, Lizard implementation does not use control statements like if() statements. So we could not get the time difference in the original message of the same dimension.

```
355    for (size_t i = 0; i < HR; ++i) {
356        uint16_t  s = (i < neg_start)?1:0;
357        uint16_t* pk_A_ri = pk_CPA.A + LWE_N * r_idx[i];
358        uint16_t* pk_B_ri = pk_CPA.B + LWE_L * r_idx[i];
359        for (int j = 0; j < LWE_N; ++j) {
360                ctx_CPA.a[j] += s * pk_A_ri[j];
361                ctx_CPA.a[j] -= (1 - s) * pk_A_ri[j];
362        }
363        for (int j = 0; j < LWE_L; ++j) {
364                ctx_CPA.b[j] += s * pk_B_ri[j];
365                ctx_CPA.b[j] -= (1 - s) * pk_B_ri[j];
366        }
367    }
```

Figure 3: Part of the encryption algorithm

However, the size of the matrix varies depending on the length of the message, so there is a difference in the amount of computation. Figure 3 shows part of the encryption algorithm. In this figure, *LWE_L* on line 363 means the dimension of the original message. Since there is no control statement in the middle, the number of calls to for() statement increases as the number of bits increases. This parameter is also used for other parts of the encryption and decryption algorithms, but it is also not affected by the control statement. For that reason we checked the computation time according to the message length. Because the measured time was very small, we used the clock_gettime() statement to represent the time up to the nano unit instead of the clock() statement. Also we used averaged values after 10,000 runs considering the sensitivity of the unit. Figure 4 shows the result. The encryption time when the message length was 10-bit was about 7.892μs and the decryption time was about 1.833μs. When the length of the message increased to 256-bit, the encryption time increased to about 14.215μs and the decryption time to about 7.510μs. The encryption time was almost doubled and the decryption time was almost quadrupled. Also, we can observe that the computation time increases almost linearly with the increase of message lengths.

However, when the length of the message is 200-bit, the encryption time is slightly reduced. We repeatedly measured the time, but still the same result

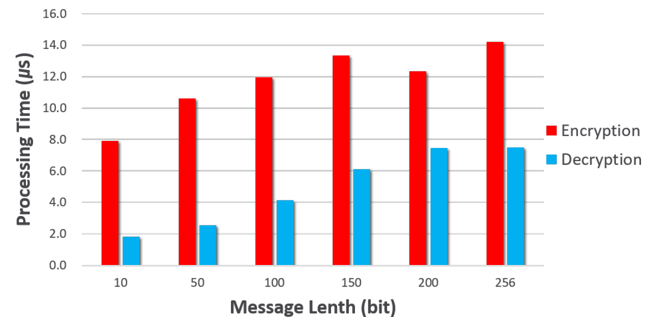

Figure 4: Encryption and Decryption times in Lizard

[1] https://github.com/LizardOpenSource/Lizard_c.git

was obtained. We tried to find the reason for this issue, but we could not find any implementation issues. We presumed that this comes from a systematic problem. In all parts of Lizard, they use a `rand()` statement that is currently considered unsafe, which may have affected the outcome. In the future work, we will replace these `rand()` statements with other secure statements and check the results accordingly.

Although there is one exception, the attacker can use the timing attack to verify that the computation time difference depends on the length of the message.

5.2 Fault Analysis

We performed the first-order fault analysis on Lizard. Unlike usual fault attacks, we performed attack on software implementation. There are three kinds of software-based fault attacks: Skipping, Zeroing, and Randomization.

> Skipping fault: It consists of skipping selected lines of the program code. It seems to be unrealistic, but it could be achieved via CPU clock glitching [26] in real world.
>
> Zeroing fault: It can be performed by setting a whole variable or a part to zero. Although it may be doubtful if attack can be realized, zeroing faults have been realized in practice [27].
>
> Randomization fault: Attacker randomly changes the value of a variable which is processing in the algorithm. The attacker benefits from knowing that it has been changed within a certain range although the value of the variable is not discovered after the attack. This attack targets the whole variable or only some bytes or bits of it [28] depending on the attacker's capabilities.

We successfully discovered secret key from skipping faults and zeroing faults on Lizard. Secret key cannot be restored from public key for randomization faults, so this part remains for future work. Detail of results is described in each section.

5.2.1 Skipping Faults

We performed the skipping faults on three parts: random number generation, addition, and modulus reduction. For skipping the random number generation, we could skip random matrix A, error matrix E, secret key S, and secret vector r.

As shown in Figure 5, line 380 is responsible for generating random matrix A. From skipping line 380, we could get plaintext from ciphertext directly. We found the fact that C code automatically set A to zero whenever we skip the generation part. Thus the result is clear since ciphertext c_1 should be zero when A equals to zero and so plaintext can be obtained from computing $m' \leftarrow \lfloor \frac{t}{p}(c_2 - S^T c_1) \rceil$. The reason for the ciphertext c_1 equal to zero is deduced from $c_1' \leftarrow A^T r$ and $c_1 \leftarrow \lfloor (\frac{p}{q}) \cdot c_1' \rceil \in Z_p^n$.

Next, we performed the skipping attacks on generation of error matrix E. Line 390 is related to generation of E as shown in Figure 6. From skipping line 390, we recovered secret key sk in a reasonable time using Gauss elimination. This result is derived from the equation $B = AS + E$. Then the plaintext pt is obtained from ciphertext ct using secret information. Recovered secret key and real secret key is shown in Figure 7. We can observe that secret keys have the same value. We executed experiment with parameter n=10, m=10, l=20 for visible results. It is certain that same result is obtained for original parameters.

```
376  void gen_A_CPA(){
377      for(int i = 0; i < LWE_M; ++i){
378          uint16_t* pk_Ai = pk_CPA.A + LWE_N * i;
379          for(int j = 0; j < LWE_N; ++j){
380              pk_Ai[j] = rand() << _16_LOG_Q;
381          }
382      }
383  }
```

Figure 5: Code for random matrix A generation

```
386  void gen_E_CPA(){
387      for(int i = 0; i < LWE_M; ++i){
388          uint16_t* pk_Bi = pk_CPA.B + LWE_L * i;
389          for(int j = 0; j < LWE_L; ++j){
390              pk_Bi[j] = SAMPLE_DG() << _16_LOG_Q;
391          }
392      }
393  }
```

Figure 6: Code for error matrix E generation

```
Our Result
0*65535*0*0*1*0*65535*0*0*0*1*65535*0*1*0*1*1*1*0*0*
65535*65535*0*65535*65535*0*1*1*1*0*0*0*1*0*0*0*1*65535*0*0*
65535*65535*1*0*0*0*1*65535*0*0*1*0*0*0*65535*65535*65535*65535*0*65535*
0*0*1*65535*1*65535*65535*0*0*0*0*0*1*0*0*1*1*0*1*
65535*65535*65535*1*0*1*1*65535*1*0*0*65535*1*1*0*65535*0*1*0*65535*
0*1*65535*65535*0*0*65535*1*0*65535*0*65535*1*0*65535*65535*1*65535*0*0*
0*0*0*1*1*0*1*0*0*0*65535*0*0*0*0*0*1*65535*0*
1*0*0*1*0*65535*65535*65535*0*0*65535*0*65535*0*65535*65535*0*0*0*1*
65535*1*0*1*1*0*0*65535*1*1*0*1*65535*0*0*1*65535*65535*0*65535*
0*1*1*65535*0*65535*1*0*65535*0*0*0*1*0*1*65535*0*0*65535*0*

Secret Key
0*65535*0*0*1*0*65535*0*0*0*1*65535*0*1*0*1*1*1*0*0*
65535*65535*0*65535*65535*0*1*1*1*0*0*0*1*0*0*0*1*65535*0*0*
65535*65535*1*0*0*0*1*65535*0*0*1*0*0*0*65535*65535*65535*65535*0*65535*
0*0*1*65535*1*65535*65535*0*0*0*0*0*1*0*0*1*1*0*1*
65535*65535*65535*1*0*1*1*65535*1*0*0*65535*1*1*0*65535*0*1*0*65535*
0*1*65535*65535*0*0*65535*1*0*65535*0*65535*1*0*65535*65535*1*65535*0*0*
0*0*0*1*1*0*1*0*0*0*65535*0*0*0*0*0*1*65535*0*
1*0*0*1*0*65535*65535*65535*0*0*65535*0*65535*0*65535*65535*0*0*0*1*
65535*1*0*1*1*0*0*65535*1*1*0*1*65535*0*0*1*65535*65535*0*65535*
0*1*1*65535*0*65535*1*0*65535*0*0*0*1*0*1*65535*0*0*65535*0*
```

Figure 7: Results for computing $S = A^{-1}B$ and secret key sk with skipping faults on error matrix E

Also skipping secret key S was performed. Line 399 is responsible for generating secret matrix S as shown in Figure 8. The result is same as skipping random matrix A since $S^T c_1$ is zero and m' is obtained from $m' \leftarrow \lfloor \frac{t}{p}(c_2 - S^T c_1) \rceil$.

Line 471 is related to generating secret vector r as shown in Figure 9. The result of skipping r is same as

```
395  void gen_sk_CPA(){
396      for(int i = 0; i < LWE_L; ++i){
397          uint16_t* sk_i = sk_CPA + LWE_N * i;
398          for(int j = 0; j < LWE_N; ++j){
399              sk_i[j] = (rand() & 0x01) + (rand() & 0x01) - 1;
400          }
401      }
402  }
```

Figure 8: Code for secret key sk generation

```
466  size_t gen_r_idx(uint16_t* r_idx){
467      size_t neg_start = 0;
468      uint64_t tmp;
469
470      for(int i = 0; i < HR/2; ++i){
471          tmp = rand();
472          neg_start += (tmp & 0x01);
473          r_idx[2 * i] = ((tmp >> 1) & 0x03ff) % LWE_M;
474          neg_start += ((tmp >> 0x7ff) & 0x01);
475          r_idx[2 * i + 1] = ((tmp >> 0xfff) & 0x03ff) % LWE_M;
476      }
477      return neg_start;
478  }
```

Figure 9: Code for secret vector r generation

skipping A or skipping S as the value of c_1 is obtained from $c_1' \leftarrow A^T r$ and $c_1 \leftarrow \lfloor (\frac{p}{q}) \cdot c_1' \rceil \in Z_p^n$.

There are three addition part in Lizard C implementation. One is in generating matrix B, another is in computing ciphertext c_2, and the other is in computing plaintext m'. Plaintext m' is recovered from skipping addition in generation of B. In contrast to result of skipping addition in B, recovering the plaintext from skipping addition in encryption phase is not possible since ciphertext c_2 stores plaintext as default value. This causes a correctness error in decryption phase. Obviously, the result is same for skipping addition in decryption phase.

Skipping the modulus reduction was not effective for recovery of plaintext. It does not reveal any secret information.

5.2.2 Zeroing Faults

We performed zeroing fault attack on random matrix A, error matrix E, secret key S, and secret vector r. All the results are same as skipping the random number generation. This is induced by the fact that C code automatically initializes variables to zero whenever skipping the generation part as mentioned earlier.

5.2.3 Randomization Faults

We performed randomization attack on secret matrix, error matrix, modulus, and randomness. It is clear that randomizing modulus and randomness is not effective since the value of the faulty modulus would remain unknown and random values are just random themselves. It was not successful to exploit vulnerability from randomizing secret matrix and error matrix. This part remains for future work.

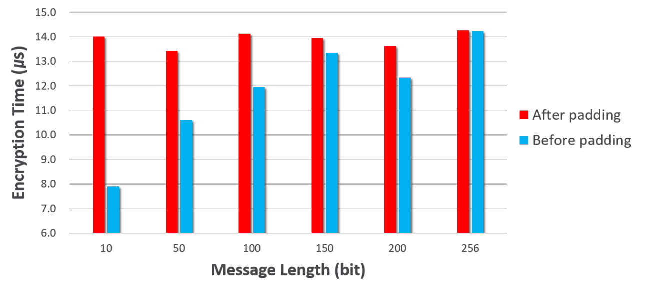

Figure 10: Comparing encryption time before and after message padding.

```
1024  uint16_t SK1[LWE_N][LWE_L];
1025    for(int k=0;k<LWE_N;++k){
1026        uint16_t* sk_k = sk_CPA + LWE_L * k;
1027          for(int j=0;j<LWE_L;++j){
1028              printf("%d ",SK1[k][j]);
1029          }
1030    }
```

Figure 11: Correctness check of secret key

6 Countermeasures

We showed the results of the timing and fault attack in Sections 4 and 5. Thus, Lizard scheme can be attacked by a side channel attack, and a proper method is needed to prevent it. So we present how to effectively defend against timing and fault attacks in this section.

6.1 Defending Timing Attacks

As mentioned in the previous section, the vulnerability of timing attacks comes from time differences in the computation process. Therefore, it is necessary to make all computation times the same to prevent such a time difference.

Message padding technique is a good method for the same computation time. In brief, this method is to add a padding bit to the original message so that the length of input message is always identical. We set the maximum input bit to 256-bit to apply this method to Lizard. Then, when the actual input is received, compare it with the maximum input bit. If this is less than the maximum input bit, add the padding bit to the difference so that it always maintains 256-bit of input. Figure 10 shows a comparison of encryption times before and after message padding. we can observe that the same encryption time appears after doing message padding. This means that the timing attack has been successfully defended.

But this technique is not perfect. If the input bit exceeds the maximum input bit, another time difference may occur. To prevent this situation, defender may increase the maximum input bit to 512 or larger. However, Lizard would become inefficient because Lizard will perform many operations on short inputs. Thus balance between security and performance is necessary.

6.2 Defending Fault Attacks

We devised a countermeasure at software source as fault analysis is performed for software implementation. Fault attacks cannot be prevented because of its feature, so we proposed the detection of these attacks. We checked the correctness of secret key for the case modifying secret matrix S as shown in Figure 11. Then we verify that it outputs zero matrix when zeroing fault attack or skipping fault attack on secret matrix is performed. As randomization fault attack does not make secret value zero, the result value is different from other attacks. Still this method is effective to detect randomization attack. Similar methods can be used for detecting attacks on random matrix A, error matrix E, and secret vector r.

To provide a long-term security, we should make an algorithm level redundancy.

7 Discussion

During our experiment, we found some results that were difficult to understand. We will suggest them in this section.

7.1 rand() Function

Lizard uses rand() function for all random number generation process. However, there is a major flaw in C rand() function. rand() function is a pseudo-random generator, so it outputs same result every time after a small number of runs. It is caused by low entropy of seed value. To counter this situation, srand() function is used to increase the entropy of seed. Still it is not suitable for random number generator used in cryptography. There are only 2^{25} possible values for the seed if an attacker knows the year in which key is generated. Then an attacker can find secret value through brute force attack in a reasonable time. In our experiment, there is no clear evidence that rand() function is vulnerable. We estimate that there is a significant weaknesses in rand() function, so it is recommended that designer use dev/urandom function for true randomness.

7.2 gen_r_idx() Function

In the gen_r_idx() function, Lizard scheme performs 0x7ff times right shifting operation for random value tmp. However, tmp is set to range from 0 to 2^{31} by the rand() function, and since 0x7ff has a value of 2047, the right shifting operation is performed 2047 times. Therefore, a value of 0 is always stored in neg_start where this operation result is stored. After that, the result of executing right shift operation 4095 times in tmp is stored in odd-numbered array of r_idx. As above, 0 is always stored in the variable. Using this, the probability that the attacker will hit the vector r is doubled. These results need to be modified appropriately because it is clear that the algorithm implementer did not intended such result. A very simple solution is to remove the shift operation.

7.3 gen_sk_CPA() Function

In the gen_sk_CPA() function, the operation to generate the secret key adds two random 0 or 1 and then subtracts 1. In this case, the probability of -1 or 1 being chosen as secret keys is 25 percent each, but the probability of 0 being chosen is 50 percent. This is an inconsistent result, making it easier for an attacker to guess the secret key value. So we need to change the output possibility of -1, 0, 1's identically.

8 Conclusion

This paper tries to find vulnerability of lattice-based encryption scheme Lizard. We performed software-based side channel attacks on Lizard including timing and fault attacks to recover secret key. For timing attack, we could not directly find secret key since Lizard does not have control statements like **if()** statements. However, we could find the time difference based on timing attack as the amount of computation depends on the size of the message input bit. This time difference can lead to various problems such as deducing the approximate length of the secret key. We also performed three types of fault attacks on Lizard; skipping, zeroing and randomization. Our attack model allows fault attack to be easily performed because we assume that the attacker can access to the source code. As a result, we successfully obtained the secret key through skipping and zeroing fault attacks, but it was not effective by randomization attack.

To prevent timing attack, we proposed a message padding technique. If the padding is added to the message input bits to maintain a constant bit, the computation time is relatively constant, so no time difference occurs. As a countermeasure for fault attacks, we need to check the correctness of secret values. This protection scheme was successful for all three fault attacks.

As future work, we will find vulnerability derived from the use of rand() function and will propose a countermeasure for this weakness. Also, we will modify gen_r_idx() and gen_sk_CPA() function in Lizard implementation to correct the errors. Finally, we will make an algorithm level redundancy for the long-term security.

Acknowledgement

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00555, Towards Provable-secure Multi-party Authenticated Key Exchange Protocol based on Lattices in a Quantum World) and National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2015R1A2A2A01006812, Design and Security Analysis of Novel Lattice-based Fully Homomorphic Signatures Robust to Quantum Computing Attack).

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings, 35th Annual Symposium on Foundations of Computer Science, FOCS'94*, pp. 124–134, IEEE, 1994.

[2] "Proposed submission requirements and evaluation criteria for the post-quantum cryptography standardization process." `http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-draft-aug-2016.pdf`.

[3] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM, JACM'09*, vol. 56, no. 6, p. 34, 2009.

[4] "IEEE P1363: Standard Specifications For Public Key Cryptography." `http://grouper.ieee.org/groups/1363/`, 2014.

[5] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky, "Lattice signatures and bimodal gaussians," in *Advances in Cryptology, CRYPTO'13*, pp. 40–56, Springer, 2013.

[6] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem," in *IEEE Symposium on Security and Privacy, IEEE S&P'15*, pp. 553–570, IEEE, 2015.

[7] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," *Advances in Cryptology, EUROCRYPT'12*, pp. 719–737, 2012.

[8] J. H. Cheon, D. Kim, J. Lee, and Y. S. Song, "Lizard: Cut off the tail!//practical post-quantum public-key encryption from LWE and LWR.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 1126, 2016.

[9] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the fortieth annual ACM symposium on Theory of computing, STOC'08*, pp. 197–206, ACM, 2008.

[10] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-Based encryption.," in *Cryptographers Track at the RSA Conference, CT-RSA'11*, vol. 6558, pp. 319–339, Springer, 2011.

[11] N. Bindel, J. Buchmann, and J. Kramer, "Lattice-based signature schemes and their sensitivity to fault attacks," in *Cryptology ePrint Archive, Report 2016/415*, 2016. http://eprint.iacr.org/2016/415.

[12] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25th USENIX Security Symposium, USENIX Security'16*, pp. 327–343, 2016.

[13] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. 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, ACM CCS'16*, pp. 1006–1018, ACM, 2016.

[14] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *International Algorithmic Number Theory Symposium, ANTS'98*, pp. 267–288, Springer, 1998.

[15] D. Coppersmith and A. Shamir, "Lattice attacks on NTRU," in *Advances in Cryptology, EUROCRYPT'97*, vol. 1233, pp. 52–61, Springer, 1997.

[16] N. Howgrave-Graham, "A hybrid lattice-reduction and meet-in-the-middle attack against NTRU," *Advances in Cryptology, CRYPTO'07*, pp. 150–169, 2007.

[17] J. H. Silverman and W. Whyte, "Timing attacks on NTRUEncrypt via variation in the number of hash calls," in *Cryptographers Track at the RSA Conference, CT-RSA'07*, pp. 208–224, Springer, 2007.

[18] A. A. Kamal and A. Youssef, "Fault analysis of the NTRUEncrypt cryptosystem," *IEICE transactions on fundamentals of electronics, communications and computer sciences, IEICE TFECCS'11*, vol. 94, no. 4, pp. 1156–1158, 2011.

[19] A. A. Kamal and A. M. Youssef, "A scan-based side channel attack on the NTRUEncrypt cryptosystem," in *Availability, Reliability and Security, 2012 Seventh International Conference on, ARES'12*, pp. 402–409, IEEE, 2012.

[20] A. Wang, X. Zheng, and Z. Wang, "Power analysis attacks and countermeasures on NTRU-based wireless body area networks," *KSII Transactions on Internet and Information Systems, TIIS'13*, vol. 7, no. 5, pp. 1094–1107, 2013.

[21] J.-E. Song, D.-G. Han, M.-K. Lee, and D.-H. Choi, "Power analysis attacks against NTRU and their countermeasures," *Journal of the Korea Institute of Information Security and Cryptology, JKIISC'09*, vol. 19, no. 2, pp. 11–21, 2009.

[22] M.-K. Lee, J. E. Song, D. Choi, and D.-G. Han, "Countermeasures against power analysis attacks for the NTRU public key cryptosystem," *IEICE transactions on fundamentals of electronics, communications and computer sciences, IEICE TFECCS'10*, vol. 93, no. 1, pp. 153–163, 2010.

[23] A. Atici, L. Batina, B. Gierlichs, and I. Verbauwhede, "Power analysis on NTRU implementations for rfids: First results," 2008.

[24] K. G. Paterson and R. Villanueva-Polanco, "Cold boot attacks on NTRU," in *International Conference in Cryptology in India, INDOCRYPT'17*, pp. 107–125, Springer, 2017.

[25] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM, CACM'09*, vol. 52, no. 5, pp. 91–98, 2009.

[26] J. Blömer, R. G. Da Silva, P. Günther, J. Krämer, and J.-P. Seifert, "A practical second-order fault attack against a real-world pairing implementation," in *Fault Diagnosis and Tolerance in Cryptography 2014 Workshop on, FDTC'14*, pp. 123–136, IEEE, 2014.

[27] P. Q. Nguyen and O. Regev, "Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT'06*, pp. 271–288, Springer, 2006.

[28] J. von Neumann, "Various techniques used in connection with random digits, paper no. 13 in monte carlo method," *NBS Applied Mathematics Series, NBS AMS'61*, no. 12, 1961.