Compact-LWE: a Public Key Encryption Scheme

Dongxi Liu, Nan Li, Jongkil Kim, and Surya Nepal

1 Introduction

In this proposal, we describe Compact-LWE, a new public key encryption scheme. Compact-LWE is also used to refer to the hardness problem on which the encryption scheme is constructed. We introduce Compact-LWE by comparing it with LWE [8].

Let $\mathbf{s}$ and $\mathbf{a}_i$ be n-dimensional vectors drawn uniformly from $\mathbb{Z}_q^n$, and let the error terms $e_i \in \mathbb{Z}_q$ be sampled from a discrete Gaussian distribution. The LWE problem involves a set of samples

$$(\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \bmod q),$$

where $\langle \mathbf{a}_i, \mathbf{s} \rangle$ denotes the inner product of $\mathbf{a}_i$ and $\mathbf{s}$. The search version of LWE problem is to recover $\mathbf{s}$ from such samples.

The Compact-LWE problem also involves a set of samples, but defined differently as

$$(\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + k * (r_i + p * e_i) \bmod q, \langle \mathbf{a}_i, \mathbf{s}' \rangle + k' * (r'_i + p * e'_i) \bmod q),$$

where $\mathbf{s}$, $\mathbf{s}'$, k, k' and p are secret values, and errors e_i, e'_i, r_i, and r'_i are uniformly sampled at random. For r_i and r'_i, they satisfy $ck * r_i + ck' * r'_i = 0 \texttt{ mod } p$, where ck and ck' are another two secret values. In addition, Compact-LWE takes short vector $\mathbf{a}_i$, consisting of entries that can be much less than an error value e_i, e'_i, r_i, or r'_i.

The security features of Compact-LWE are briefly discussed here, with more details given later. The well-known lattice-based attacks to LWE [5, 6, 4] are based on efficiently solving the hard Closest Vector Problem (CVP) in lattice. Even if k and k' are correctly guessed, Compact-LWE is resistant to such attacks, because the errors in a sample ($r_i + p * e_i$ or $r'_i + p * e'_i$) can be much bigger than the entries of $\mathbf{a}_i$ and thus CVP does not apply.

Another hard problem in lattices is the Short Integer Solution (SIS) problem. If SIS can be efficiently solved, the solution may help the attacker guess k, k', p, ck, and ck'. With the correct guess of these secret values, the attack can remove r_i and r'_i and generate a new sample that includes the error value $ck * e_i + ck' * e'_i$ (i.e., errors are always two-dimensional in Compact-LWE). Since the error value $ck * e_i + ck' * e'_i$ is still big, the lattice-based attacks cannot apply even if SIS problem is efficiently solved for Compact-LWE.

Hence, even if the hard problems in lattice, such as CVP and SIS, can be efficiently solved, the secret values or private key in Compact-LWE still cannot be efficiently recovered. This allows Compact-LWE to choose very small dimension parameters, such as $n = 8$ in our experiment.

2 Specification of the Compact-LWE Public Key Encryption Scheme

The specification describes the public parameters of the scheme and its three algorithms for key generation, encryption, and decryption.

2.1 Public Parameters

The scheme is specified by nine public parameters: $q, t, n, m, w, w', b, b', l$, which are all positive integers, with q being the largest one. Let pp denote the public parameters. The meaning of the parameters will be introduced when they are used. The public parameters should satisfy the following basic conditions, with other conditions to be introduced later with the corresponding algorithms.

- $m > n + 2$
- $w > w'$
- $(w - w') * b' > t$
- $l \geq 3$
- t is a power of two integer.

In the following algorithms, all random numbers are uniformly sampled.

2.2 Key Generation

Given the public parameters, the key generation algorithm generates a random key pair $(\mathbf{SK}, \mathbf{PK})$, where $\mathbf{SK}$ is the private key and $\mathbf{PK}$ is the public key. Let $\mathbf{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$ denote the key generation algorithm.

The key generation algorithm depends on four private parameters: sk_max, p_size, e_min, and e_max. The conditions $e\_max > e\_min$ and $e\_min * w > w' * (e\_max + 1)$ apply to e_min and e_max.

2.2.1 Private Key A private key is a tuple $(\mathbf{s}, k, sk, ck, \mathbf{s}', k', sk', ck', p)$, generated in the following steps.

- $\mathbf{s}$ and $\mathbf{s}'$ each is a n-dimensional vector, randomly sampled from $\mathbb{Z}_q^n$.
- k and k' are uniformly sampled from $\mathbb{Z}_q$ and must be coprime with q.
- p is randomly sampled from the set $\{(w + w') * b', ..., (w + w') * b' + p\_size\}$, satisfying
  - coprime with q, and
  - $sk\_max * b' + p + e\_max * p < q/(w + w')$.
- ck and ck' are uniformly sampled from $\mathbb{Z}_p$ and one of them must be coprime with p. In this specification, we assume ck' is coprime with p.
- sk and sk' are uniformly sampled from $\mathbb{Z}_{sk\_max}$, with $sk * ck + sk' * ck'$ coprime with p.

2.2.2 Public Key

After the private key $\mathbf{SK}$ is generated, the algorithm $\mathbf{gen}(pp)$ then generates the corresponding public key $\mathbf{PK}$. The public key $\mathbf{PK}$ consists of m random Compact-LWE samples, as defined below.

Let $\mathbf{a}_i \in \mathbb{Z}_b^n$ be a vector uniformly sampled for the ith public key sample, and $u_i \in \mathbb{Z}_{b'}$, $e_i \in [e\_min, e\_max]$, and $e'_i \in [e\_min, e\_max]$ be three random integers. Another two randomly sampled integers $r_i \in \mathbb{Z}_p$, and $r'_i \in \mathbb{Z}_p$ satisfies

$$ck * r_i + ck' * r'_i = 0 \texttt{ mod } p.$$

Since ck' is coprime with p, the above condition can always be satisfied. Let $k_q * k_q^{-1} = 1 \texttt{ mod } q$, and $k'_q * {k'}_q^{-1} = 1 \texttt{ mod } q$. Then, the ith public key sample is the tuple $(\mathbf{a}_i, u_i, pk_i, pk'_i)$, where

- $pk_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + k_q^{-1} * (sk * u_i + r_i + e_i * p) \texttt{ mod } q$, and
- $pk'_i = \langle \mathbf{a}_i, \mathbf{s}' \rangle + {k'}_q^{-1} * (sk' * u_i + r'_i + e'_i * p) \texttt{ mod } q$.

2.3 Encryption

The encryption algorithm consists of two parts: basic encryption and general encryption. Basic encryption is only able to encrypt messages in $\mathbb{Z}_t$, while general encryption can encrypt messages of any length and encodes them before calling the basic encryption algorithm.

2.3.1 Basic Encryption

A plaintext value v for basic encryption comes from $\mathbb{Z}_t$. It is encrypted into a ciphertext $\mathbf{c}$ with the public key $\mathbf{PK}$, denoted $\mathbf{c} = \mathbf{enc}(\mathbf{PK}, v)$. The ciphertext $\mathbf{c}$ is a $(n+3)$-dimensional vector. The basic encryption algorithm needs to generate an m-dimensional vector $\mathbf{l}$. Let $\mathbf{l}[i]$ indicate its ith element and its first element is $\mathbf{l}[1]$. The XOR operation is denoted by $\oplus$ and the opertation $\texttt{rol}(u, d)$ circuar shifts a value u by d bits to the left. The basic encryption algorithm works by the following steps.

- Generate the m-dimensional random vector $\mathbf{l}$, such that
  - $w \leq \Sigma_{i=1}^{m} \mathbf{l}[i] \leq w + w'$ for all $\mathbf{l}[i] > 0$ (i.e., the sum of all positive entries of $\mathbf{l}$ is a random value in between w and $w + w'$),
  - $-w' \leq \Sigma_{i=1}^{m} \mathbf{l}[i] \leq 0$ for all $\mathbf{l}[i] < 0$ (i.e., the sum of all negative entries of $\mathbf{l}$ is a random value in between $-w'$ and 0), and
  - $\Sigma_{i=1}^{m} \mathbf{l}[i] * u_i > 0$.
- Generate the ciphertext

  $$\mathbf{c} = (\Sigma_{i=1}^{m} \mathbf{l}[i] * \mathbf{a}_i, f(v, \Sigma_{i=1}^{m} \mathbf{l}[i] * u_i), \Sigma_{i=1}^{m} \mathbf{l}[i] * pk_i \texttt{ mod } q, \Sigma_{i=1}^{m} \mathbf{l}[i] * pk'_i \texttt{ mod } q),$$

  where

  $$f(v, \Sigma_{i=1}^{m} \mathbf{l}[i] * u_i) = (v \oplus \texttt{rol}(u, \log_2(t)/2)) * u' \texttt{ mod } t,$$

  $$u = (\Sigma_{i=1}^{m} \mathbf{l}[i] * u_i) \texttt{ mod } t, \text{and}$$

  $$u' \geq (\Sigma_{i=1}^{m} \mathbf{l}[i] * u_i)/t \text{ is the smallest integer coprime with t.}$$

2.3.2 General Encryption General encryption encodes and then encrypts messages. OAEP or other padding schemes can be used. However, we use a way that is easier to implement.

Suppose I is an array of 256 random bytes. Let $I[i]$ denote the ith entry of I. Similarly, given a message m, $m[i]$ denotes its ith byte. The algorithm $\mathbf{encode}(I, m) = m'$ encodes m into m'. Suppose the length of m is $\texttt{len}(\mathrm{m})$ bytes. Then, the padding has

$$pl = \log_2(t) * \lceil 8 * (\texttt{len}(m) + l)/\log_2(t)\rceil/8 - \texttt{len}(m)$$

bytes. The following is the pseudocode of $\mathbf{encode}$.

```
– Append 0xFF to $m$, followed by $pl - 3$ bytes each having the value $pl$.
– Generate two random bytes $r$ and $r'$.
– $x = I[r]$ and $r\_ori = r$
– for $i$=1 to $\texttt{len}(m) + pl - 2$ do
–     $m'[i] = x \oplus m[i]$
–     $x = x \oplus I[(m'[i] + r\_ori) \texttt{ mod } 256]$
–     $r = r \oplus ((m'[i] * r') \texttt{ mod } 256)$
– $x = I[r']$ and $r\_ori = r'$
– for $i = \texttt{len}(m) + pl - 2$ to 1 do
–     $m'[i] = x \oplus m'[i]$
–     $x = I[(m'[i] + r\_ori) \texttt{ mod } 256]$
–     $r' = r' \oplus ((m'[i] * r) \texttt{ mod } 256)$
– $m'[\texttt{len}(\mathrm{m}) + pl - 1] = r$
– $m'[\texttt{len}(\mathrm{m}) + pl] = r'$
```

After encoding, the general encryption algorithm divides a long message into blocks, each of which has $\log_2(t)$ bytes, and encrypts each block with the basic encryption algorithm.

2.4 Decryption

The decryption algorithm also consists of basic decryption and general decryption. Basic decryption only decrypts cipher blocks generated from basic encryption, while general decryption can decrypt ciphertexts of any length.

2.4.1 Basic Decryption Let $\mathbf{SK} = (\mathbf{s}, k, sk, ck, \mathbf{s}', k', sk', ck', p)$ be the private key. Given the ciphertext $\mathbf{c} = (\mathbf{a}, d, pk, pk')$, the basic decryption algorithm $\mathbf{dec}(\mathbf{SK}, \mathbf{c}) = v$ recovers the plaintext value v with the following steps.

- Calculate $d_1 = (pk - \langle \mathbf{a}, \mathbf{s}\rangle) * k \texttt{ mod } q$, and $d_1' = (pk' - \langle \mathbf{a}, \mathbf{s}'\rangle) * k' \texttt{ mod } q$.
- Let $d_2 = ck * d_1 + ck' * d_1' \texttt{ mod } p$.
- Calculate $d_3 = sckInv * d_2 \texttt{ mod } p$, where $sckInv$ is determined by $sckInv * (sk * ck + sk' * ck') = 1 \texttt{ mod } p$.
- Obtain $v = f^{-1}(d, d_3)$, where

$$f^{-1}(d, d_3) = (u'^{-1}_p * d \texttt{ mod } t) \oplus \texttt{rol}(u, \log_2(t)/2),$$

$$u = d_3 \texttt{ mod } t,$$

$$u' \geq d_3/t \text{ is the smallest integer coprime with t, and}$$

$$u'^{-1}_p * u' = 1 \texttt{ mod } t.$$

2.4.2 General Decryption Given a byte array of ciphertext, general decryption divides it into blocks, decrypts each block by using the basic decryption algorithm, and after all blocks are decrypted, decodes the messages. If the padding does not match the byte 0xFF followed by $pl - 3$ bytes each having the value pl, then general decryption returns a failure.

General decryption uses the algorithm $\mathbf{decode}(I, m') = m$ to decode m' into the plaintext m. The decoding algorithm is defined with the following pseudocode.

```
– Let $l = \texttt{len}(m')$, $r' = m'[l]$, $r = m'[l-1]$
– for $i = 1$ to $l - 2$ do
–     $r' = ((m'[i] * r) \texttt{ mod } 256) \oplus r'$
– $x = I[r']$
– for $i = l - 2$ to 1 do
–     $m[i] = x \oplus m'[i]$
–     $x = I[(m'[i] + r') \texttt{ mod } 256]$
– for $i = 1$ to $l - 2$ do
–     $r = ((m[i] * r') \texttt{ mod } 256) \oplus r$
– $x = I[r]$
– for $i$=1 to $l - 2$ do
–     $y = m[i]$
–     $m[i] = x \oplus m[i]$
–     $x = I[(y + r) \texttt{ mod } 256]$
– Check padding as described above.
```

Any changes to m' leads to the change to x in each loop and the random numbers r and r'. Then, the changes are propagated to every byte of m.

2.5 Correctness

We discuss the correctness of basic encryption and basic decryption algorithms. Our basic decryption algorithm is deterministically correct, as analyzed below. At the first step of decryption, we have

$$d_1 = sk * \Sigma_{i=1}^{m}\mathbf{l}[i] * u_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * r_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * e_i * p \texttt{ mod } q$$

and

$$d_1' = sk' * \Sigma_{i=1}^{m}\mathbf{l}[i] * u_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * r_i' + \Sigma_{i=1}^{m}\mathbf{l}[i] * e_i' * p \texttt{ mod } q.$$

Due to the condition on $\mathbf{l}$ and $sk\_max * b' + p + e\_max * p < q/(w + w')$, we know

$$0 < sk * \Sigma_{i=1}^{m}\mathbf{l}[i] * u_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * r_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * e_i * p < q,$$

implying that

$$d_1 = sk * \Sigma_{i=1}^{m}\mathbf{l}[i] * u_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * r_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * e_i * p.$$

Similarly, we have

$$d_1' = sk' * \Sigma_{i=1}^{m}\mathbf{l}[i] * u_i + \Sigma_{i=1}^{m}\mathbf{l}[i] * r_i' + \Sigma_{i=1}^{m}\mathbf{l}[i] * e_i' * p.$$

Then, at the second step of decryption, since $ck * r_i + ck' * r'_i = 0 \ \mathtt{mod}\ p$, we obtain

$$d_2 = (ck * sk + ck' * sk') * \Sigma_{i=1}^{m} \mathbf{l}[i] * u_i \ \mathtt{mod}\ p.$$

Due to the condition $(w + w') * b' < p$, we know $\mathbf{l}[i] * u_i < p$ and $\mathbf{l}[i] * u_i$ is then recovered at the third step, leading to the same u and u' as used in basic encryption.

3 Security and Attacks

3.1 Hardness and IND-CCA2 Security

As introduced in Section 1, compared with LWE, a Compact-LWE sample includes extra secret values ($\mathbf{s}'$, k, k', p, ck, and ck') and extra errors (r_i, r'_i, and e'_i), and extra public parameters. Informally, given a LWE sample, by choosing $k = 1$, $p = 1$, and any values for other secrets and errors, the adversary can convert a LWE sample into a Compact-LWE sample, in which the extra public parameter b takes the same value as q. Hence, if the adversary can find $\mathbf{s}$ from Compact-LWE samples, then the same algorithm can be used to find $\mathbf{s}$ from LWE samples. Since it is hard to solve the search LWE problem [8], it is also hard to recover secret value $\mathbf{s}$ from Compact-LWE samples. Similarly, $\mathbf{s}'$ is also hard from Compact-LWE samples. Without knowing $\mathbf{s}$ and $\mathbf{s}'$, the adversary cannot determine the values of $k*(r_i+p*e_i) \bmod q$ and $k'*(r'_i+p*e'_i) \bmod q$ in each Compact-LWE sample, making the recovery of secret values (k, k', p, ck, and ck') hard. In the next section, with concrete attacks, we will show when the extra public parameter b takes a value smaller enough than q, the search of the original secrets in both LWE and Compact-LWE is even harder.

Our scheme can achieve IND-CCA2 security. Informally, when a ciphertext is changed (or adaptively chosen) by the adversary, the basic decryption algorithm adds a random error of at least ${u'}_p^{-1}$ to the encoded message. The error is then propagated to every byte of the decoded message by the decoding algorithm. When an padding, which is at least l bytes, does not match, the general decryption algorithm just returns failure, indicated by -1 in implementation.

3.2 Attacks to Public Keys

This type of attacks aims to recover private keys from the corresponding public keys. These attacks have been developed to attack LWE, including algebraic attacks [2], combinatorial attacks [1, 3], and lattice-based attacks [5, 6, 4].

Since the number of samples in our public keys is limited, algebraic attacks and combinatorial attacks are not effective [7]. We discuss more on lattice-based attacks. We first introduce lattice-based attacks to LWE.

Suppose there are m LWE samples. Let $\mathbf{A}$ be a $n * m$ matrix constructed by taking each of the m vectors $\mathbf{a}_i$ as a column of $\mathbf{A}$, and let $\mathbf{e}$ be a m-dimensional error vector obtained by collecting e_i as its entries. Then, the lattice-based attacks to LWE try to find $\mathbf{s}$ from the m samples by solving CVP in lattices [5, 6, 4]. That is, in the lattice generated from the row vectors of $\mathbf{A}$, the problem is to find a lattice point that is closest to the target $\mathbf{A}^T\mathbf{s} + \mathbf{e}$. In LWE, the vector $\mathbf{e}$ has a small Euclidean norm $||\mathbf{e}||$, and thus the lattice point closest to $\mathbf{A}^T\mathbf{s} + \mathbf{e}$ is $\mathbf{A}^T\mathbf{s}$, which can then be used to recover $\mathbf{s}$ by solving a system of noiseless linear equations.

In Compact-LWE, the errors $k * (r_i + p * e_i)$ (or $k' * (r_i' + p * e_i')$) can be as big as q, apparently making CVP not applicable to Compact-LWE. If k and k' have been correctly guessed, the errors become $r_i + p * e_i$ (or $r_i' + p * e_i'$), which are much bigger than elements in $\mathbf{a}_i$, making $\mathbf{A}^T\mathbf{s}+\mathbf{e}$ and $\mathbf{A}^T\mathbf{s}$ not closest to each other. Thus, even if k and k' have been correctly guessed, lattice-based attacks [5, 6, 4] are still not applicable to Compact-LWE, even not applicable to LWE with short $\mathbf{a}_i$. With the tool provided in [4], Figure 1 shows that lattice-based attacks fail to find $\mathbf{s}$ from LWE samples when b becomes small enough (i.e., short $\mathbf{a}_i$). In the experiment, for each b, 50 sets of LWE samples are generated and attacked, with $n = 9$, $m = 24$, $q = 2^{48}$, and e_i is uniformly sampled from $\mathbb{Z}_{3200}$ or $\mathbb{Z}_{1600}$.

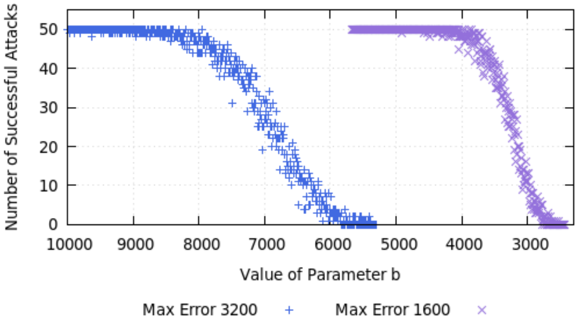

Fig. 1. Resistance to Lattice-based Attacks

For example, if the first element of $\mathbf{s}$ is increased by 1, the changed secret still can satisfy each LWE sample by changing the original e_i into $e_i - \mathbf{a}_i[1]$, where $\mathbf{a}_i[1]$ is the first element of $\mathbf{a}_i$. When e_i has a bigger range than $\mathbf{a}_i[1]$, $e_i - \mathbf{a}_i[1]$ can still be a valid error. Similarly, for Compact-LWE, when k is correctly guessed, if the first element of $\mathbf{s}$ is increased by k, r_i can be changed into $r_i - \mathbf{a}_i[1]$ to counteract the change to $\mathbf{s}$. Thus, when $\mathbf{a}_i$ is short enough and the dimension n is very small, the lattice-based tool can quickly return a secret, which however is not the original secret $\mathbf{s}$. For Compact-LWE encryption scheme, when r_i is changed, the condition $ck * r_i + ck * r_i = 0 \bmod p$ is no longer valid and the decryption will fail.

SIS is another hard lattice problem [7]. If SIS can be easily solved, the attacker can reduce Compact-LWE samples to the samples like $k_q^{-1} * (r_{sis} + p * e_{sis}) \bmod q$ (and $k'^{-1}_q * (r'_{sis} + p * e'_{sis}) \bmod q$), from which the attacker can determine k_q^{-1} by correctly guessing p, r_{sis}, and e_{sis}. Note that in $k_q^{-1} * (r_{sis} + p * e_{sis}) \bmod q$, r_{sis} can be bigger than p and much bigger than e_{sis}. Hence, p and r_{sis} cannot be uniquely guessed. For example, $p - 1$ and $r_{sis} + e_{sis}$ are also a valid guess.

If both r_{sis} and r'_{sis} happen to be correctly guessed, a pair of valid ck and ck' can be determined, since $ck * r_{sis} + ck' * r'_{sis} = 0 \bmod p$. Note that the attacker can choose $ck = 1$ and determine the value ck' according to $ck * r_{sis} + ck' * r'_{sis} = 0 \bmod p$ (or the other way).

After k_q^{-1}, ${k'}_q^{-1}$, p, and a valid pair of ck and ck' are correctly guessed, the attacker can remove r and r' in the public sample by rewriting a public key sample $(\mathbf{a}_i, u_i, pk_i, pk'_i)$ into the following one:

$$(\mathbf{a}_i * k_q * ck, \mathbf{a}_i * {k'}_q * ck', u_i, pk''_i),$$

where

- $pk''_i = pk_i * k_q * ck + pk'_i * k'_q * ck' \bmod q$
  $= (sk * ck + sk' * ck') * u_i + \langle \mathbf{a}_i * k_q * ck, \mathbf{s} \rangle + \langle \mathbf{a}_i * k'_q * ck', \mathbf{s}' \rangle + (ck * e_i + ck' * e'_i + y_i) * p \bmod q$,
- $y_i \in \mathbb{Z}_p$ is determined by $y_i = (ck * r_i + ck' * r'_i)/p$.

The new sample above includes the error $ck * e_i + ck' * e'_i + y_i$, which is still big. If the attacker insists on applying lattice-based attacks to the rewritten public key, the original secrets $\mathbf{s}$, $\mathbf{s}'$ and $ck * sk + ck'sk'$ cannot be recovered as illustrated in Figure 1. Since $ck * e_i + ck' * e'_i + y_i$ can be much bigger than $2 * e\_max$, the wrong secrets obtained in such attacks will generate too much error in the result of the first step of basic decryption (i.e., d_1 and d'_1). Too much error violates the correctness condition $(sk\_max * b' + p + e\_max * p < q/(w + w'))$, hence failing to generate correct results.

Let $ck = 1$. The attacker thus has to exhaustively search correct e'_i for at least $2n+1$ new samples and then deals with the smaller remaining error $e_i + y_i$ by applying lattice-based attacks to recover $\mathbf{s}$, $\mathbf{s}'$, $ck * sk + ck' * sk'$ from a public key. Note that $e_i + y_i$ is still bigger than elements in $\mathbf{a}_i$ or even u_i; hence, a correct guess of e'_i still cannot guarantee the recovery of valid $\mathbf{s}$, $\mathbf{s}'$, $ck * sk + ck' * sk'$.

3.3 Attacks to Ciphertexts

Without knowing the private key, an attacker may try to recover a message from the ciphertext and the public key. Since a long message is divided into blocks to encrypt, the attacker needs to correctly guess $\mathbf{l}$ for each message block. Due to padding, there are at least $\lceil 8 * l/\log_2(t) \rceil$ blocks for a message and the decoding algorithm ensures that even one bit change can be propagated to every byte of the decoded message and returns -1 if padding is not correct. Hence, the guess to $\mathbf{l}$ must be correct for all blocks at the same time.

For a single ciphertext block, since $m > n + 2$, there can be a large number of solutions for $\mathbf{l}$ by choosing big enough m for the public key and big enough w and w' for basic encryption. Note that the size of a public key increases with m, but not the size of ciphertexts.

Each solution for $\mathbf{l}$ can lead to a decrypted message block. When the number of the solutions of $\mathbf{l}$ is as big as t, this type of attacks does not make sense for the attacker. For example, if $t = 2$ and the solutions of $\mathbf{l}$ allows both 0 and 1 to be decrypted from a ciphertext block, then the attacker cannot determine which is the correct plaintext by attacking ciphertexts.

4 Performance Analysis and Evaluation

The performance of key generation algorithm and basic encryption/decryption algorithms is analyzed below. The analysis relies on the parameters q, t, n, m, w, w', b, b'.

4.1 Performance of Key Generation

A private key is defined as $(\mathbf{s}, k, sk, ck, \mathbf{s}', k', sk', ck', p)$, which includes $2*(n+3)+1$ random numbers that need to be randomly sampled and checked for the corresponding conditions. In particular, sk, sk', and ck' might need to be generated a few more times to ensure ck' is coprime with p and $sk*ck+sk'*ck'$ is also coprime with p. In addition, our implementation chooses to compute the multiplicative inverses of k, k', ck', and $sk*ck+sk'*ck'$, and stores these multiplicative inverses as part of the private key. Hence, they will not be computed during the generation of public keys and decryption,

A public key consists of m samples. For each sample, the inner product of two n-dimensional vectors needs to be calculated twice, three random numbers (r_i, e_i, e'_i) need to be sampled, plus a small constant number of additions and multiplications. Asymptotically, the time complexity of key generation algorithm is $O(n)$ for generating private keys and $O(m*n)$ for generating public keys.

Recall that encoding and decoding algorithms rely on the byte array I, which is hard-coded in our implementation. To reduce the size of public keys, the vector $\mathbf{a}_i \in \mathbb{Z}_b^n$ is extracted from I deterministically. To avoiding storing u_i, the first public sample uses the first element of I as u_1, and lets $u_i = pk_{i-1} \oplus pk'_{i-1} \ \mathtt{mod}\ b'$ for $1 < i \leq m$. Hence, in addition to seven public parameters (i.e., q, t, n, m, w, w'), a public key consists of m pairs of pk_i and pk'_i, each of which has $\log_2(q)$ bits.

4.2 Performance of Basic Encryption

The encryption algorithm first generates an m-dimensional vector $\mathbf{l}$. The sum of its positive entries is at most $w+w'$ and the sum of its negative entries is at least $-w'$. Hence, at most $w+2*w'$ random number needs to be sampled to generate $\mathbf{l}$. Moreover, if the sum of the negative entries is too big (reflected by the third condition on $\mathbf{l}$), another at most w' operations are need to reduce some negative entries until the third condition on $\mathbf{l}$ holds. Hence, the time complexity of generating $\mathbf{l}$ is $O(w+3*w')$.

After $\mathbf{l}$ is generated, the algorithm multiplies each of m vectors in the public key with the corresponding entry $\mathbf{l}$, and then all m resulting vectors are added. The complexity of this step in encryption is $O(m*(n+3))$. Hence, the complexity of basic encryption is $O(w+3*w') + O(m*(n+3))$.

A ciphertext from basic encryption is an $(n+3)$-dimensional vector. The size of the first n elements is at most $\log_2((w+w')*b)$ bits, the following one has $log_2(t)$ bits, and the last two both have $\log_2(q)$ bits. Hence, the total size of a ciphertext block from basic encryption is

$$n*\log_2((w+w')*b) + log_2(t) + 2*\log_2(q)$$

bits. In our implementation, a ciphertext block is a little bit longer since each of the first n elements is byte-aligned. The padding in general encryption adds at most $\lceil 8*l/\log_2(t)\rceil$ message blocks to encrypt.

4.3 Performance of Basic Decryption

The decryption algorithm needs to calculate the inner product of two n-dimensional vectors twice, plus a constant number of extra multiplications and additions. Hence, the time complexity is $O(2n)$.

5 Advantages and Limitations

Compact-LWE has the following advantages:

- Compact-LWE is constructed on simple mathematical objects (only integers and modular arithmetic over integers), which thus is easy to be understood and thoroughly analyzed by the cryptographic community.
- Even if the hardness problems in lattices (such as CVP and SIS) can be solved efficiently, the security of Compact-LWE are not affected as illustrated and analyzed in Section 3.2. Hence, the dimension parameter n in Compact-LWE can be very small and Compact-LWE can generate short ciphertexts.
- The selection of parameters is straightforward. All secret values and errors are sampled uniformly at random. For parameters satisfying the conditions specified in the scheme, the scheme does not have any decryption failures, as proved in Section 2.5.
- The simplicity of Compact-LWE makes it easy to implement. In addition, it can be used as a lightweight public encryption scheme. It has an implementation for Contiki Operating System, with evaluation over Tmote Sky wireless sensor nodes.
- Key generation in Compact-LWE is efficient. When used in key-establishment protocols, Compact-LWE can support forward secrecy by generating fresh key pairs for each new session.

Compared with RSA and ECC, a limitation of Compact-LWE is that the size of its public keys is bigger. For example, in the experiment described below, the public key is about 2K bytes.

6 Performance Evaluation

6.1 Platform Description

The performance is evaluated on 64-bit Ubuntu 17.04 running on Window 7 with VMware Workstation 12 Player. The computer is Dell Latitude E7470, with Intel Core i5 CPU and 8GB RAM.

6.2 Parameter Selection

Table 1 and Table 2 list the parameters to be used in the evaluation and to be used to define the security strength category.

q	t	n	m	w	w'	b	b'	l
2^{64}	2^{32}	8	128	224	32	16	68719476736	8

Table 1. Public Parameters

sk_max	p_size	e_min	e_max
229119	16777216	457	3200

Table 2. Private Parameters

6.3 Security Strength Category

As discussed in Section 3.2, even if k_q^{-1}, ${k'}_q^{-1}$, p, ck, and ck' can be correctly guessed, the attacker still has to exhaustively search correct e_i (or e'_i) for at least $2n + 1$ samples to recover secrets $\mathbf{s}$, $\mathbf{s}'$, $ck * sk + ck' * sk'$ from a public key. Recall that $e\_min \leq e_i, e'_i \leq e\_max$. Given the above parameter configuration, the search space is at least

$$\log_2(e\_max - e\_min) * (2n + 1) = \log_2(3200 - 457) * (2 * 8 + 1) = 194$$

bits, which is comparable to the search space of AES192. Hence, any attacks that recover a security key from a public key of Compact-LWE must require computational resources comparable to those required for key search on AES192.

6.4 Sizes of Keys and Ciphertexts

For the above configuration, the private key and the public key are 232 bytes and 2064 bytes, respectively. We use 2 bytes to store each of the first 8 elements; hence, the basic encryption generates ciphertexts of 36 bytes for a 4-byte message block. For messages of different lengths, the sizes of corresponding ciphertexts are given in Table 3. As an example, for a 32-byte message and 8-byte padding, it is divided into 10 blocks and the general encryption returns a ciphertext of 360 bytes.

Message (bytes)	32	64	128	256	512	1024
Ciphertext (bytes)	360	648	1224	2376	4680	9288

Table 3. Ciphertext Size

6.5 Efficiency

The performance is based on the optimised implmentation of the scheme, which is compiled with -O2 option of gcc.

The time of generating 10000 key pairs is about 1.55 seconds. The performance of encryption and decryption depends on the length of messages, as shown in Table 4.

Message (bytes)	32	64	128	256	512	1024
Encryption (seconds)	1.29	2.15	4.36	7.56	14.81	28.78
Decryption (seconds)	0.18	0.27	0.43	0.88	1.78	3.50

Table 4. Performance of 10000 Encryptions and Decryptions

References

1. Albrecht, M.R., Cid, C., Faugère, J.C., Fitzpatrick, R., Perret, L.: On the complexity of the bkw algorithm on lwe. Des. Codes Cryptography 74(2), 325–354 (Feb 2015)
2. Arora, S., Ge, R.: New algorithms for learning in presence of errors. In: Proceedings of the 38th International Colloquim Conference on Automata, Languages and Programming - Volume Part I. pp. 403–415. ICALP'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2027127.2027170
3. Kirchner, P., Fouque, P.: An improved BKW algorithm for LWE with applications to cryptography and lattices. In: Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I. pp. 43–62 (2015)
4. Kirshanova, E., May, A., Wiemer, F.: Parallel implementation of BDD enumeration for LWE. In: Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings. pp. 580–591 (2016)
5. Lindner, R., Peikert, C.: Better key sizes (and attacks) for lwe-based encryption. In: Kiayias, A. (ed.) Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6558, pp. 319–339. Springer (2011)
6. Liu, M., Nguyen, P.Q.: Solving bdd by enumeration: An update. In: Proceedings of the 13th International Conference on Topics in Cryptology. pp. 293–309. CT-RSA'13 (2013)
7. Micciancio, D., Peikert, C.: Hardness of SIS and LWE with small parameters. In: Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. pp. 21–39 (2013)
8. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005. pp. 84–93. ACM (2005)