Compact-LWE-MQ^{H} : Public Key Encryption without Hardness Assumptions

Dongxi Liu* Surya Nepal

CSIRO Data61, Australia

Abstract. Modern public key encryption relies on various hardness assumptions for its security. Hardness assumptions may cause security uncertainty, for instance, when a hardness problem is no longer hard or the best solution to a hard problem might not be publicly released.
In this paper, we propose a public key encryption scheme Compact-LWE-MQ^{H} to demonstrate the feasibility of designing public key encryption without relying on hardness assumptions. Instead, its security is based on problems that are called factually hard.The two factually hard problems we proposed in this work are stratified system of linear and quadratic equations, and learning with relatively big errors. Such factually hard problems have the structures to ensure that they can only be solved by exhaustively searching their solution spaces, even when the problem size is very small.
Based on the structure of factually hard problems, we prove that without brute-force search the adversary cannot recover plaintexts or private key components, and then discuss CPA-security and CCA-security of Compact-LWE-MQ^{H} . We have implemented Compact-LWE-MQ^{H} in SageMath. In a configuration for 128-bit security level, the public key has 3708 bytes and a ciphertext is around 574 bytes.

1 Introduction

Public key encryption is usually constructed around various assumptions of computationally hard problems as its security foundation. The well-known hardness assumptions, for instance, include the problems of factoring the product of big prime numbers, computing discrete logarithms over finite fields, multivariate quadratic problems [10], and the problems in lattice, such as short integer solution problem and shortest vector problem [1].

When there are new algorithms that can improve the efficiency of solving hard problems, the relevant public key encryption schemes need to increase their key length as an counter measure. However, if there is a completely new computing paradigm, such as quantum computing, some hardness assumptions no longer hold and hence new public key encryptions have to be designed, as being done by NIST Post-Quantum Cryptography Standardization project.

In this paper, we investigate the feasibility of designing public key encryption based on hardness facts, rather than assumptions. For this purpose, we design the public key encryption scheme Compact-LWE-MQ^{H} , which has its security based on the following two hardness problems that we argue are factually hard.

* Corresponding author: dongxi.liu@csiro.au

1.1 Factual Hardness Problems

For each hardness problem described below, we analyze the necessary operations to recover involved secret values in the problem. If such necessary operations involve brute-force search, this problem is factually hard. A factually hard problem still needs to be solved with brute-force search even when it is configured with small parameters.

Let h, h', and p be three prime numbers, and m, n, a_max three integers. We require $q > m * p * (h' + h)$, $h > m * p * p$, $h' > m * p * p$, $m > n$, and $p > a\_max$.

1.1.1 Learning with Relatively Big Errors Let $\mathbf{s}$ and $\mathbf{s}'$ be uniformly sampled at random from $\mathbb{Z}_q^n$. For $0 \leq i \leq m-1$, we can have m samples:

$$(a_i, a'_i, b_i = \langle \mathbf{s}, \mathbf{a}_i \rangle + r_i \bmod q, b'_i = \langle \mathbf{s}', \mathbf{a'}_i \rangle + r'_i \bmod q)_i$$

where r_i and r'_i are uniform from $\mathbb{Z}_h$ and $\mathbb{Z}_{h'}$, respectively, $\mathbf{a}_i$ and $\mathbf{a'}_i$ uniformly sampled from $\mathbb{Z}_{a\_max}^n$. In the samples, $\mathbf{s}$, $\mathbf{s}'$, r_i, r'_i, h and h' are kept secret.

Compared with Learning with Errors (LWE) [9], the elements in $\mathbf{a}_i$ and $\mathbf{a'}_i$ here can be much smaller than r_i and r'_i. Thus, given proper parameters (m, p, and a_max), the solutions to the noised equations below (i.e., $\mathbf{s}$ and $\mathbf{s}'$) will not necessarily be unique.

$$b_i \approx \langle \mathbf{s}, \mathbf{a}_i \rangle \bmod q,\ b'_i \approx \langle \mathbf{s}', \mathbf{a'}_i \rangle \bmod q$$

This fact can be verified with experiments. For example, since $\mathbf{a}_i[0]$ is small, $r_i + \mathbf{a}_i[0]$ can be a value in $\mathbb{Z}_h$ for $0 \leq i \leq m-1$, and thus $\tilde{\mathbf{s}}$ is a valid solution with a high chance, where $\tilde{\mathbf{s}}[0] = \mathbf{s}[0] - 1$ and $\tilde{\mathbf{s}}[j] = \mathbf{s}[j]$ for $1 \leq j \leq n-1$. Note that it is not a fact to LWE [9].

Hence, to recover the error terms r_i and r'_i originally used in each sample, the valid solution space of $\mathbf{s}$ and $\mathbf{s}'$ have to be searched. Each valid solution of $\mathbf{s}$ and $\mathbf{s}'$ leads to the possible values of r_i and r'_i. In Compact-LWE-MQ$^{\mathtt{H}}$, the error terms r_i and r'_i are further multiplied with secret values from $\mathbb{Z}_q$ before adding them to b_i and b'_i, respectively.

Note that this problem is not a NP problem, because it is easy find a solution to the above noised equations, though the solution might be different from $\mathbf{s}$ and $\mathbf{s}'$.

1.1.2 Stratified System of Linear and Quadratic Equations Let x_i $(0 \leq i \leq m-1)$ and v all be random numbers uniformly sampled from $\mathbb{Z}_p$, and let b_i and b'_i be defined as above. Then, we define the following linear equations, where the values of cb, cb', b_i, and b'_i are known, with x_i, x'_i, and v regarded as unknowns,

$$cb = \Sigma_{i=0}^{m-1} x_i * b_i \bmod q$$
$$cb' = \Sigma_{i=0}^{m-1} x'_i * b'_i \bmod q$$

and m multivariate quadratic (MQ) equations,

$$x'_i = v * x_i \bmod p.$$

The adversary aims to recover v from the above equations. Note that v only appears in the m quadratic equations. To recover v without guessing it, x_i and x'_i must be known for $0 \leq i \leq m-1$ and their values must lead to consist v, since each pair of x_i and x'_i can determine a value for v.

We call the system with above equations stratified, because the values of x_i and x'_i can only be calculated from the linear equations, and then these values are used to solve the quadratic equations to determine the value of v. The method of solving a linear system for the values of some variables and then determining the values of other variables in quadratic terms is also taken by all existing algorithms to solve multivariate quadratic (MQ) equations, such as [3, 4].

The linear equations in our stratified system are underdetermined, thus allowing a number of solutions to x_i and x'_i. As a result, recovering v in this stratified system is factually hard, since each solution to x_i and x'_i has to be checked. Moreover, v is generated into a vector in Compact-LWE-MQ$^{\mathtt{H}}$.

1.2 Overview of Compact-LWE-MQ$^{\mathtt{H}}$

Compact-LWE-MQ$^{\mathtt{H}}$ is constructed around the above hardness facts and with them enhanced. Briefly, the first hardness fact ensures the private key cannot be recovered from the public key, and the second prevents efficient attacks of recovering plaintexts from ciphertexts and the public key.

In addition to $\mathbf{s}$ and $\mathbf{s}'$ as described above, the private key of Compact-LWE-MQ$^{\mathtt{H}}$ includes a number of extra secret components (i.e., h, h', and others), formed as a hidden layer of learning with relatively big errors samples, for improving its resilience to attacks. Without knowing extra secrets, the adversary can find many solutions to $\mathbf{s}$ and $\mathbf{s}'$, but no way to know which solution is useful for an attack. On the other hand, only correct $\mathbf{s}$ and $\mathbf{s}'$ can lead to the recovery of the extra secrets in the hidden layer. It is a deadlock for the attacker.

The plaintext in Compact-LWE-MQ$^{\mathtt{H}}$ is a n'-dimensional vector $\mathbf{v} \in \mathbf{Z}_p^{n'}$. The minimum of n' is determined by the expected security level and distribution of plaintexts. Compact-LWE-MQ$^{\mathtt{H}}$ achieves CPA-security when $\mathbf{v}$ is random enough for the adversary. After being slightly revised, Compact-LWE-MQ$^{\mathtt{H}}$ is also CCA-Secure.

The security of Compact-LWE-MQ$^{\mathtt{H}}$ is not proved by hardness reduction. Instead, in the proof, we identify the necessary steps or operations that have to be taken by the adversary to manipulate the public keys or ciphertexts. For example, a step to solve a system of MQ equations is to find the solution of a linear system [3, 4]. The design of Compact-LWE-MQ$^{\mathtt{H}}$ requires the adversary has to do brute-force search at these necessary steps.

Due to this security proof method, Compact-LWE-MQ$^{\mathtt{H}}$ has to have a simple construction; otherwise, there might be too many ways for the adversary to start an attack, hence not easy to correctly identity necessary attack steps. For this purpose, Compact-LWE-MQ$^{\mathtt{H}}$ is constructed with only basic modular arithmetic operations, as illustrated above in the definitions of the factually hard problems. This simplicity also means that it can be analyzed by people with different expertise. Hence, if there is a design flaw, it can be easily discovered in a short period of time by different groups of people; otherwise, there will be no attacks other than the brute-force ones in any case.

Compact-LWE-MQ$^{\mathtt{H}}$ is straightforward to implement due to its simple construction. We have implemented Compact-LWE-MQ$^{\mathtt{H}}$ in SageMath and the implementation is in Appendix. For a configuration of 128-bit security, the scale of samples is very small, with $m = 24$ and $n = 4$, and p is around 128 bits. In the configuration, the size of the public key is 3708-byte and a ciphertext is 574-byte long. Note that performance is not a focus of Compact-LWE-MQ$^{\mathtt{H}}$ in this paper.

1.3 Notations

Given a positive integer q, $\mathbb{Z}_q$ refers to the set $\{0, ..., q-1\}$. The n-ary Cartesian product of $\mathbb{Z}_q$ is denoted by $\mathbb{Z}_q^n$. For a finite set, e.g. S, $x \leftarrow S$ means that x is uniformly sampled from S at random.

A lower-case boldface letter denotes a vector or a list (e.g., $\mathbf{s}$), while an upper-case boldface letter (e.g., $\mathbf{L}$) is used to represent a matrix (or a list of lists). Given two vectors with the same dimension, such as $\mathbf{a}$ and $\mathbf{s}$, their inner product is denoted by $\langle \mathbf{a}, \mathbf{s} \rangle$.

The ith element of list $\mathbf{a}$ is written as $\mathbf{a}[i]$ and i starts from 0. The sublist of list $\mathbf{a}$ from its ith element to $(j-1)$th element, inclusively, is denoted by $\mathbf{a}[i:j]$. If $j \leq i$, then $\mathbf{a}[i]$ returns an empty list []. Two vectors $\mathbf{a}$ and $\mathbf{b}$ are added by $\mathbf{a}+\mathbf{b}$. The concatenation of two lists $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a}\|\mathbf{b}$. An element x is appended to the tail of list $\mathbf{a}$ by $\mathbf{a}+x$. The length of list $\mathbf{a}$ is denoted by $\mathtt{len}(\mathbf{a})$. The prime number just after an integer n is denoted by $\mathtt{np}(n)$. $\mathbf{0}$ means a zero vector, with its dimension derived from its usage context.

2 Construction of Compact-LWE-MQ$^{\mathtt{H}}$

As a public-key encryption scheme, Compact-LWE-MQ$^{\mathtt{H}}$ is defined with three algorithms: key generation, encryption, and decryption. Compact-LWE-MQ$^{\mathtt{H}}$ is parameterized with four integers: n, m, p, and a_max. We require p be a prime, $a\_max < p$, $n < m < p$. Another integer parameter q will be generated by the key generation algorithm. These public parameters are represented by pp. They are taken as implicit inputs by the encryption and decryption algorithms.

2.1 Key Generation

The key generation algorithm generates the private key $\mathbf{SK}$ and the public key $\mathbf{PK}$, denoted by $\mathbf{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$. The key generation algorithm is comprised of private key generation and public key generation.

2.1.1 Private Key A private key $\mathbf{SK}$ is a tuple $(\mathbf{s}, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k, \mathbf{s}', \mathbf{k}', \mathbf{t}', \mathbf{z}', s', h', k')$, generated in the steps below.

- $h = \mathtt{np}(m * p * p + r)$, where $r \leftarrow \mathbb{Z}_p$.
- $h' = \mathtt{np}(m * p * p + r')$, where $r' \leftarrow \mathbb{Z}_p$.
- $q = m * p * (h + h') + q'$, where $q' \leftarrow \mathbb{Z}_p$.
- $\mathbf{s} \leftarrow \mathbb{Z}_q^n$, $\mathbf{k} \leftarrow \mathbb{Z}_p^n$, $\mathbf{t} \leftarrow \mathbb{Z}_p^n$, $\mathbf{s}' \leftarrow \mathbb{Z}_q^n$, $\mathbf{k}' \leftarrow \mathbb{Z}_p^n$, and $\mathbf{t}' \leftarrow \mathbb{Z}_p^n$.

– $\mathbf{z} \leftarrow \mathbb{Z}_h^n$, and $\mathbf{z}' \leftarrow \mathbb{Z}_{h'}^n$.
– $s \leftarrow \mathbb{Z}_q$ and $s' \leftarrow \mathbb{Z}_q$, ensuring
  • s and s are co-prime with q.
– $k \leftarrow \mathbb{Z}_p$, $k' \leftarrow \mathbb{Z}_p$, ensuring
  • $k \neq 0$, and $k' \neq 0$.

Note that q generated above is a public parameter. All secret values in $\mathbf{SK}$ and other random values used below are randomly sampled from uniform distributions. Given $\mathbf{SK}$, we write $\widehat{\mathbf{SK}}$ for $(\mathbf{s}', \mathbf{k}', \mathbf{t}', \mathbf{z}', s', h', k', \mathbf{s}, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k)$.

2.1.2 Public Key Based on the private key $\mathbf{SK}$, the algorithm $\mathbf{gen}$ generates the corresponding public key $\mathbf{PK}$, which is comprised of m samples. The ith sample, denoted $(\mathbf{a}_i, b_i, \mathbf{a}'_i, b'_i)$, is defined as:

– $\mathbf{a}_i \leftarrow \mathbb{Z}_{a\_max}^n$ and $\mathbf{a}'_i \leftarrow \mathbb{Z}_{a\_max}^n$.
– $u_i \leftarrow \mathbb{Z}_p$.
– $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + s * r_i \bmod q$, where
  • $r_i = (\langle \mathbf{a}_i, \mathbf{k} \rangle + \langle \mathbf{a'}_i, \mathbf{t} \rangle + k * u_i \bmod p) + \langle \mathbf{a}_i, \mathbf{z} \rangle \bmod h$.
– $b'_i = \langle \mathbf{a'}_i, \mathbf{s}' \rangle + s' * r'_i \bmod q$, where
  • $r'_i = (\langle \mathbf{a'}_i, \mathbf{k}' \rangle + \langle \mathbf{a}_i, \mathbf{t}' \rangle + k' * u_i \bmod p) + \langle \mathbf{a'}_i, \mathbf{z}' \rangle \bmod h'$.

As shown above, $\mathbf{PK}$ is defined by embedding one layer of Learning with Relatively Big Error samples into another. The top layer is $(\mathbf{a}_i, \mathbf{a'}_i, b_i, b'_i)_i$, and the bottom layer is $(\mathbf{a}_i, \mathbf{a'}_i, r_i, r'_i)_i$. The bottom layer is a hidden layer, in terms that the moduli (i.e., h and h'), r_i, and r'_i are not known publicly. Without guessing the hidden values (h, h', r_i, and r'_i), the adversary does not have any way to recover $\mathbf{z}$ and $\mathbf{z}'$.

The notation $\widehat{\mathbf{PK}}$ is used in in the encryption algorithm below. It has the same number of samples as $\mathbf{PK}$, and for each $(\mathbf{a}, b, \mathbf{a}', b') \in \mathbf{PK}$, $(\mathbf{a}', b', \mathbf{a}, b) \in \widehat{\mathbf{PK}}$.

2.2 Encryption

A plaintext $\mathbf{v}$ in Compact-LWE-MQ$^{\mathtt{H}}$ is an element in $\mathbb{Z}_p^{n'}$, where n' is an odd integer if bigger than two. The minimum of n' is relevant to the expected security level and the size of p. There is no upper bound on n'. That is, the size of $\mathbf{v}$ can be incremented for longer messages. We will discuss the choice of n' later when we analyze the security of Compact-LWE-MQ$^{\mathtt{H}}$.

The encryption algorithm $\mathbf{enc}$ is defined in Algorithm 1. With the public key $\mathbf{PK}$, this algorithm encrypts a message $\mathbf{v} \in \mathbb{Z}_p^{n'}$ into a ciphertext $\mathbf{c}$, denoted $\mathbf{enc}(\mathbf{PK}, \mathbf{v}) = \mathbf{c}$. The ciphertext $\mathbf{c}$ is comprised of n' components, each of which is generated with the algorithm $\mathbf{encS}$ in Algorithm 2.

In the $\mathbf{enc}$ algorithm, there are n' vectors each randomly sampled from $\mathbb{Z}_p^m$ and stored in $\mathbf{L}$, which is then passed as a parameter to $\mathbf{encS}$. The algorithm $\mathbf{encS}$ is

Algorithm 1: **enc** for encryption

input : $\mathbf{PK}$, $\mathbf{v}$
output: $\mathbf{c}$

Let $\mathbf{L}$ be a list with n' entries.
for $i = 0$ **to** $n' - 1$ **do**
  $\mathbf{L}[i] \leftarrow \mathbb{Z}_p^m$
end
$\mathbf{c} = []$
for $i = 0$ **to** $n' - 1$ **do**
  if $i \bmod 2 = 0$ **then**
    $c = \mathtt{encS}(\mathbf{PK}, \mathbf{L}, \mathbf{v}, i)$
  end
  else
    $c = \mathtt{encS}(\widehat{\mathbf{PK}}, \mathbf{L}, \mathbf{v}, i)$
  end
  $\mathbf{c} = \mathbf{c} + c$
end
return $\mathbf{c}$

Algorithm 2: **encS** for encrypting ciphertext component

input : $\mathbf{PK}', \mathbf{L}, \mathbf{v}, i$
output: c

$\mathbf{l} = \mathbf{L}[i]$
$\mathbf{l}' = \mathbf{0}$
for $j = 0$ **to** $n' - 1$ **do**
  $\mathbf{l}' = \mathbf{l}' + \mathbf{v}[j] * \mathbf{L}[(i + j) \bmod n'] \bmod p$
end
$\mathbf{ca}_1 = \mathbf{0}, cb_1 = 0, \mathbf{ca}_2 = \mathbf{0}, cb_2 = 0$
for $j = 0$ **to** $m - 1$ **do**
  Let $(\mathbf{a}_1, b_1, \mathbf{a}_2, b_2) = \mathbf{PK}'[j]$.
  $\mathbf{ca}_1 = \mathbf{ca}_1 + \mathbf{l}[j] * \mathbf{a}_1$
  $cb_1 = cb_1 + \mathbf{l}[j] * b_1 \bmod q$
  $\mathbf{ca}_2 = \mathbf{ca}_2 + \mathbf{l}'[j] * \mathbf{a}_2$
  $cb_2 = cb_2 + \mathbf{l}'[j] * b_2 \bmod q$
end
$c = (\mathbf{ca}_1, cb_1, \mathbf{ca}_2, cb_2)$
return c

invoked n' times; at the ith time ($0 \leq i \leq n'-1$), the ith ciphertext component c is generated and appended to the ciphertext $\mathbf{c}$.

In the $\mathbf{encS}$ algorithm, two vectors $\mathbf{l}$ and $\mathbf{l}'$ are derived from $\mathbf{L}$: $\mathbf{l}$ is simply $\mathbf{L}[i]$ and $\mathbf{l}'$ is obtained by adding vectors $\mathbf{L}[(i+j) \bmod n'] * \mathbf{v}[j]$ from $j=0$ to $j=n'-1$ over modulus p. Then, each sample in $\mathbf{PK}'$ is multiplied by the corresponding elements in $\mathbf{l}$ and $\mathbf{l}'$, and all such samples are summed up as the ciphertext component $c = (\mathbf{ca}_1, cb_1, \mathbf{ca}_2, cb_2)$.

Depending on whether n' is even or odd, the public key passed to $\mathbf{encS}$ is different. Particularly, $\mathbf{PK}'$ is $\mathbf{PK}$ when n' is even, and $\widehat{\mathbf{PK}}$ otherwise. Suppose $n'=2$. Then, let the two ciphertext components in $\mathbf{c}$ be $c_0 = (\mathbf{ca}_1, cb_1, \mathbf{ca}_2, cb_2)$ and $c_1 = (\mathbf{ca}'_1, cb'_1, \mathbf{ca}'_2, cb'_2)$. Since $\mathbf{PK}$ and $\widehat{\mathbf{PK}}$ are used for c_0 and c_1, respectively, cb_1 and cb'_1 (or cb_2 and cb'_2) cannot be homomorphically added together; otherwise, values from $\mathbf{Z}_h$ and $\mathbf{Z}_{h'}$ are mixed.

Algorithm 3: $\mathbf{dec}$ for decryption

input : $\mathbf{SK}$, $\mathbf{c}$
output: $\mathbf{v}$ or -1

$n' = \texttt{len}(\mathbf{c})$
$\mathbf{g} = [], \mathbf{y} = []$
for $i = 0$ **to** $n'-1$ **do**
  if $i \bmod 2 = 0$ **then**
    $g, y = \texttt{decS}(\mathbf{SK}, \mathbf{c}[i])$
  end
  else
    $g, y = \texttt{decS}(\widehat{\mathbf{SK}}, \mathbf{c}[i])$
  end
  $\mathbf{g} = \mathbf{g} + g$
  $\mathbf{y} = \mathbf{y} + y$
end
Let $\mathbf{G}$ be a $n' * n'$ matrix.
for $i = 0$ **to** $n'-1$ **do**
  $\mathbf{G}[i] = \mathbf{g}[i:n'] \| \mathbf{g}[0:i]$
end
if $\mathbf{G}$ *is singular with repect to* p **then**
  return -1
end
$\mathbf{v} = \mathbf{G}^{-1} * \mathbf{y} \bmod p$
return $\mathbf{v}$

When $\mathbf{v}$ is a zero vector, each ciphertext component in $\mathbf{c}$ contains zero for the last two elements. For example, $\mathbf{v} = \mathbf{0}$ leads to $\mathbf{ca}_2 = \mathbf{0}$ and $\mathbf{cb}_2 = 0$ in the above ciphertext component. Hence, before being passed to the encryption algorithm, $\mathbf{v}$ should be processed to ensure it is not a zero vector.

2.3 Decryption

The decryption algorithm $\mathbf{dec}$ is defined in Algorithm 3. With the private key $\mathbf{SK}$, the decryption algorithm recovers the value $\mathbf{v}$ from ciphertext $\mathbf{c}$ if $\mathbf{c}$ can be decrypted, denoted $\mathbf{dec}(\mathbf{SK}, \mathbf{c}) = \mathbf{v}$; otherwise, the decryption algorithm returns -1.

The ciphertext $\mathbf{c}$ is a list, consisting of ciphertext components. Let the length of $\mathbf{c}$ be n'(i.e., n' ciphertext components contained in $\mathbf{c}$). For each ciphertext component $\mathbf{c}[i]$, the $\mathbf{dec}$ algorithm invokes the algorithm $\mathbf{decS}$, defined in Algorithm 4, to return a pair of integers g and y in $\mathbf{Z}_p$, which are then appended to two lists $\mathbf{g}$ and $\mathbf{y}$, respectively. Note that when i is even, $\mathbf{SK}$ is passed to $\mathbf{decS}$; otherwise, $\widehat{\mathbf{SK}}$ is used, corresponding to $\mathbf{PK}$ and $\widehat{\mathbf{PK}}$ in encryption.

After processing all ciphertext components, $\mathbf{g}$ and $\mathbf{y}$ each contains n' elements. From $\mathbf{g}$, the $\mathbf{dec}$ algorithm derives the $n' * n'$ matrix $\mathbf{G}$, which contains $\mathbf{g}$ as the first row and left rotations of $\mathbf{g}$ as the remaining rows, left rotating by one element each time for $n' - 1$ times.

If $\mathbf{G}$ is invertible with respect to p, the plaintext $\mathbf{v}$ is recovered by $\mathbf{G}^{-1} * \mathbf{y}$, where $\mathbf{G}^{-1}$ is the inverse of $\mathbf{G}$ with respect to modulus p. Otherwise, $\mathbf{c}$ cannot be decrypted and the decryption algorithm returns -1.

Algorithm 4: $\mathbf{decS}$ for decrypting ciphertext component

input : $\mathbf{SK}', (\mathbf{ca}_1, cb_1, \mathbf{ca}_2, cb_2)$
output: g, y

Let $\mathbf{SK}' = (\mathbf{s}_1, \mathbf{k}_1, \mathbf{t}_1, \mathbf{z}_1, s_1, h_1, k_1, \mathbf{s}_2, \mathbf{k}_2, \mathbf{t}_2, \mathbf{z}_2, s_2, h_2, k_2)$.
Let $s_1^{-1} \in \mathbb{Z}_{q_1}$, such that $s_1^{-1} * s_1 = 1 \bmod q$.
Let $s_2^{-1} \in \mathbb{Z}_{q_2}$, such that $s_2^{-1} * s_2 = 1 \bmod q$.
Let $k_1^{-1} \in \mathbb{Z}_p$, such that $k_1^{-1} * k_1 = 1 \bmod p$.
Let $k_2^{-1} \in \mathbb{Z}_p$, such that $k_2^{-1} * k_2 = 1 \bmod p$.
$d = s_1^{-1} * (cb_1 - \langle \mathbf{ca}_1, \mathbf{s}_1 \rangle) \bmod q$
$d = (d - \langle \mathbf{ca}_1, \mathbf{z}_1 \rangle) \bmod h_1$
$g = k_1^{-1} * (d - \langle \mathbf{ca}_1, \mathbf{k}_1 \rangle) + \langle \mathbf{ca}_1, \mathbf{t}_2 * k_2^{-1} \rangle \bmod p$
$d' = s_2^{-1} * (cb_2 - \langle \mathbf{ca}_2, \mathbf{s}_2 \rangle) \bmod q$
$d' = (d' - \langle \mathbf{ca}_2, \mathbf{z}_2 \rangle) \bmod h_2$
$y = k_2^{-1} * (d - \langle \mathbf{ca}_2, \mathbf{k}_2 \rangle) + \langle \mathbf{ca}_2, \mathbf{t}_1 * k_1^{-1} \rangle \bmod p$
return g, y

In Algorithm 4, we need the multiplicative inverses of s_1, s_2, k_1, and k_2. Their existence is guaranteed, since they are either co-prime with q or required not being 0 in private key generation.

2.4 Correctness

Compact-LWE-MQ$^{\mathtt{H}}$ is deterministically correct if the ciphertext can be decrypted. A cipherext cannot be decrypted when the determinant of $\mathbf{G}$ modulo p is zero. Hence, the probability of failing to decrypt is $\frac{1}{p}$.

Theorem 1 (Correctness). *Let* $\mathbf{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$, $\mathbf{v} \in \mathbb{Z}_p^{n'}$ *and* $\mathbf{c} = \mathbf{enc}(\mathbf{PK}, \mathbf{v})$. *Then,* $\mathbf{dec}(\mathbf{SK}, \mathbf{c}) = \mathbf{v}$ *with probability* $\frac{p-1}{p}$, *or* $\mathbf{dec}(\mathbf{SK}, \mathbf{c}) = -1$ *with probability* $\frac{1}{p}$.

Proof. For brevity, we prove the case where $n' = 2$. Let $(\mathbf{a}_i, b_i, \mathbf{a}'_i, b'_i) = \mathbf{PK}[i]$ for $0 \leq i \leq m-1$, and let $\mathbf{l}_1 = \mathbf{L}[0]$, $\mathbf{l}'_1 = \mathbf{v}[0] * \mathbf{L}[0] + \mathbf{v}[1] * \mathbf{L}[1] \bmod p$, $\mathbf{l}_2 = \mathbf{L}[1]$, and $\mathbf{l}'_2 = \mathbf{v}[0] * \mathbf{L}[1] + \mathbf{v}[1] * \mathbf{L}[0] \bmod p$, where $\mathbf{L}$ is supposed to be sampled in encryption.

When $n' = 2$, $\mathbf{c}$ contains two ciphertext components: $(\mathbf{ca}_1, cb_1, \mathbf{ca}'_1, cb'_1)$, and $(\mathbf{ca}_2, cb_2, \mathbf{ca}'_2, cb'_2)$, where

$$
\begin{aligned}
\mathbf{ca}_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * \mathbf{a}_i \\
cb_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * b_i \bmod q \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * (\langle \mathbf{a}_i, \mathbf{s} \rangle + s * r_i) \bmod q \\
\mathbf{ca}'_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * \mathbf{a}'_i \\
cb'_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * b'_i \bmod q \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * (\langle \mathbf{a}'_i, \mathbf{s}' \rangle + s' * r'_i) \bmod q \\
\mathbf{ca}_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * \mathbf{a}'_i \\
cb_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * b'_i \bmod q \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * (\langle \mathbf{a}'_i, \mathbf{s}' \rangle + s' * r'_i) \bmod q \\
\mathbf{ca}'_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * \mathbf{a}_i \\
cb'_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_2[i] * b_i \bmod q \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}'_2[i] * (\langle \mathbf{a}_i, \mathbf{s} \rangle + s * r_i) \bmod q
\end{aligned}
$$

Due to $m * p * (h + h') < q$, $m * p * p < h$, and $m * p * p < h'$, in Algorithm 4, we obtain

$$
\begin{aligned}
d_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * r_i \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * (\langle \mathbf{a}_i, \mathbf{k} \rangle + \langle \mathbf{a}'_i, \mathbf{t} \rangle + k * u_i) \bmod p \\
d'_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * r'_i \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * (\langle \mathbf{a}'_i, \mathbf{k}' \rangle + \langle \mathbf{a}_i, \mathbf{t}' \rangle + k' * u_i) \bmod p \\
d_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * r'_i \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * (\langle \mathbf{a}'_i, \mathbf{k}' \rangle + \langle \mathbf{a}_i, \mathbf{t}' \rangle + k' * u_i) \bmod p \\
d'_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_2[i] * r_i \\
&= \Sigma_{i=0}^{m-1} \mathbf{l}'_2[i] * (\langle \mathbf{a}_i, \mathbf{k} \rangle + \langle \mathbf{a}'_i, \mathbf{t} \rangle + k * u_i) \bmod p
\end{aligned}
$$

Still in Algorithm 4, from d_1, d'_1, d_2, and d_2, the following values can be recovered.

$$
\begin{aligned}
g_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}_1[i] * (\langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1} + u_i + \langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1}) \bmod p \\
y_1 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_1[i] * (\langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1} + u_i + \langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1}) \bmod p \\
&= \Sigma_{i=0}^{m-1} (\mathbf{v}[0] * \mathbf{l}_1 + \mathbf{v}[1] * \mathbf{l}_2)[i] * (\langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1} + u_i + \langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1}) \bmod p \\
g_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}_2[i] * (\langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1} + u_i + \langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1}) \bmod p \\
y_2 &= \Sigma_{i=0}^{m-1} \mathbf{l}'_2[i] * (\langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1} + u_i + \langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1}) \bmod p \\
&= \Sigma_{i=0}^{m-1} (\mathbf{v}[0] * \mathbf{l}_2 + \mathbf{v}[1] * \mathbf{l}_1)[i] * (\langle \mathbf{a}'_i, \mathbf{t} \rangle * k^{-1} + u_i + \langle \mathbf{a}_i, \mathbf{t}' \rangle * k'^{-1}) \bmod p
\end{aligned}
$$

Thus, we have $y_1 = \mathbf{v}[0] * g_1 + \mathbf{v}[1] * g_2 \bmod p$, and $y_2 = \mathbf{v}[0] * g_2 + \mathbf{v}[1] * g_1 \bmod p$. Hence, if the matrix $\mathbf{G} = \begin{bmatrix} g_1 & g_2 \\ g_2 & g_1 \end{bmatrix}$ is invertible with respect to p, the message $\mathbf{v}$ can be correctly recovered. The same proof applies when $n' > 2$; the only difference is that $\mathbf{G}$ becomes a bigger $n' * n'$ matrix. □

3 Security Analysis

The security of Compact-LWE-MQ$^{\mathtt{H}}$ is not based on computational hardness assumption, so the reduction-based proof methodology cannot be applied. Instead, we prove its security by analyzing the necessary steps the adversary has to take in order to derive plaintexts from ciphertexts or to derive secret key components from the public key. Our proof will show that brute-force search is needed in the identified necessary steps. Note that we only identify the necessary steps, not all steps for a complete attack.

3.1 Attack to Ciphertexts

In this attack, the adversary tries to recover the plaintext $\mathbf{v}$ from the ciphertext $\mathbf{c}$ and the public key $\mathbf{PK}$.

Theorem 2. *Let* $\mathbf{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$*, and* $\mathbf{c} = \mathbf{enc}(\mathbf{PK}, \mathbf{v})$*, where* $\mathbf{v} \in \mathbb{Z}_p^{n'}$*. Then, from* $\mathbf{PK}$ *and* $\mathbf{c}$*,* $\mathbf{v}$ *can only be recovered by exhaustive search.*

Proof. For brevity, we take $n' = 2$. When $n' = 2$, $\mathbf{c}$ contains two ciphertext components: $(\mathbf{ca}_1, cb_1, \mathbf{ca}'_1, cb'_1)$, and $(\mathbf{ca}_2, cb_2, \mathbf{ca}'_2, cb'_2)$. For $0 \leq i \leq m-1$, let x_{1i}, ${x'}_{1i}$, x_{2i}, and ${x'}_{2i}$ be $4 * m$ variables.

Then, from the ciphertext $\mathbf{c}$ and the pubic key $\mathbf{PK}$, the adversary can obtain a stratified system, consisting of the following linear equations,

$$\mathbf{ca}_1 = \Sigma_{i=0}^{m-1} x_{1i} * \mathbf{a}_i \tag{1}$$

$$cb_1 = \Sigma_{i=0}^{m-1} x_{1i} * b_i \bmod q \tag{2}$$

$$\mathbf{ca}'_1 = \Sigma_{i=0}^{m-1} {x'}_{1i} * \mathbf{a}'_i \tag{3}$$

$$cb'_1 = \Sigma_{i=0}^{m-1} {x'}_{1i} * b'_i \bmod q \tag{4}$$

$$\mathbf{ca}_2 = \Sigma_{i=0}^{m-1} x_{2i} * \mathbf{a}'_i \tag{5}$$

$$cb_2 = \Sigma_{i=0}^{m-1} x_{2i} * b'_i \bmod q \tag{6}$$

$$\mathbf{ca}'_2 = \Sigma_{i=0}^{m-1} {x'}_{2i} * \mathbf{a}_i \tag{7}$$

$$cb'_2 = \Sigma_{i=0}^{m-1} {x'}_{2i} * b_i \bmod q \tag{8}$$

and the following quadratic equations ($0 \leq i \leq m-1$), where $\mathbf{v}[0]$ and $\mathbf{v}[1]$ are also regarded as unknowns,

$${x'}_{1i} = \mathbf{v}[0] * x_{1i} + \mathbf{v}[1] * x_{2i} \bmod p \tag{9}$$

$${x'}_{2i} = \mathbf{v}[0] * x_{2i} + \mathbf{v}[1] * x_{1i} \bmod p. \tag{10}$$

The variables $\mathbf{v}[0]$ and $\mathbf{v}[1]$ appear only in multivariate quadratic terms in equations (9) and (10). Hence, the values of $\mathbf{v}[0]$ and $\mathbf{v}[1]$ can only be determined from (9) and (10) by first knowing the values of x_{1i}, ${x'}_{1i}$, x_{2i}, and ${x'}_{2i}$. Moreover, for each $0 \leq i \leq m-1$, x_{1i}, ${x'}_{1i}$, x_{2i}, and ${x'}_{2i}$ must lead to the same $\mathbf{v}[0]$ and the same $\mathbf{v}[1]$ for consistency.

The values of x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} can only be determined by solving linear equations (1)-(8). Based on the condition on public parameters, there can be a large number of solutions for equations (1)-(8). The adversary thus has to search the solution space of equations (1)-(8) for each x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} to find consistent $\mathbf{v}[0]$ and $\mathbf{v}[1]$.

On the other hand, for $0 \leq i \leq m-1$, by multiplying $\mathbf{a}'_i$ (or $\mathbf{a}_i$) at both sides of equations (9) (or equations (10)) and summing each side, the adversary can have $\Sigma_{i=0}^{m-1} x'_{1i} * \mathbf{a}'_i = \mathbf{ca}'_1$ and $\Sigma_{i=0}^{m-1} x_{2i} * \mathbf{a}'_i = \mathbf{ca}_2$ (or $\Sigma_{i=0}^{m-1} x'_{2i} * \mathbf{a}_i = \mathbf{ca}'_2$ and $\Sigma_{i=0}^{m-1} x_{1i} * \mathbf{a}_i = \mathbf{ca}_1$). Thus, equations (9) and (10) are reduced to:

$$\mathbf{ca}'_1 = \mathbf{v}[0] * (\Sigma_{i=0}^{m-1} x_{1i} * \mathbf{a}'_i) + \mathbf{v}[1] * \mathbf{ca}_2 \bmod p \tag{11}$$

$$\mathbf{ca}'_2 = \mathbf{v}[0] * (\Sigma_{i=0}^{m-1} x_{2i} * \mathbf{a}_i) + \mathbf{v}[1] * \mathbf{ca}_1 \bmod p. \tag{12}$$

In the equations (11) and (12), the values of $\Sigma_{i=0}^{m-1} x_{1i} * \mathbf{a}'_i$ and $\Sigma_{i=0}^{m-1} x_{2i} * \mathbf{a}_i$ are not known. Hence, in order to to recover $\mathbf{v}_0$ and $\mathbf{v}_1$ from (11) and (12), the adversary still needs to search the solution spaces of equations (1)-(2) and (5)-(6) for possible values of x_{1i} and x_{2i}.

At last, the adversary can search each pair of $\mathbf{v}[0]$ and $\mathbf{v}[1]$ from $\mathbb{Z}_p^2$, respectively. If the search space is small, for instance when $\mathbf{v}$ is not uniform, this search step is feasible. By guessing $\mathbf{v}[0]$ and $\mathbf{v}[1]$, equations (1) to (10) become a linear system. If this linear system allows x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} to have absolute values around p, then the adversary knows both $\mathbf{v}[0]$ and $\mathbf{v}[1]$ are correctly guessed. □

With the implementation of Compact-LWE-MQ$^{\mathtt{H}}$, we have done experiments to confirm the solution space of equations (1)-(2) (or other equations from (3) to (8)) and the attacks by guessing $\mathbf{v}[0]$ and $\mathbf{v}[1]$. A correct guess leads to values of x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} smaller than those in incorrect guesses.

3.2 Attack to Public Keys

The aim of the adversary in this attack is to recover the components in the secret key $\mathbf{SK}$ from the public key $\mathbf{PK}$. In the sample of $\mathbf{PK}$, the error term r_i (or r'_i) is multiplied by $s \in \mathbf{Z}_q$ (or $s' \in \mathbf{Z}_q$). Hence, the adversary has to know s and s' before performing any lattice-based attacks to recover $\mathbf{s}$ and $\mathbf{s}'$.

One way to achieve this purpose is that the adversary generates a short m-dimensional vector, denoted $\mathbf{l}$, such that $\Sigma_{i=1}^{m}(\mathbf{l}[i] * \mathbf{a}_i) = \mathbf{0}$ (and/or $\Sigma_{i=1}^{m}(\mathbf{l}[i] * \mathbf{a}'_i) = \mathbf{0}$).Then, the adversary can get $\Sigma_{i=1}^{m}(\mathbf{l}[i] * b_i) = s * \Sigma_{i=1}^{m}(\mathbf{l}[i] * r_i) \bmod q$. Then, the adversary can guess $\Sigma_{i=1}^{m}(\mathbf{l}[i] * r_i)$, which is much smaller than s.

In the parameter configuration proposed later, the size of p is longer than 128 bits. Thus, guessing $\Sigma_{i=1}^{m}(\mathbf{l}[i] * r_i)$ still needs to search in a big space. However, even if s and s' have been somehow correctly guessed, the following theorem guarantees the security of $\mathbf{SK}$ with respect to the size of p.

Theorem 3. *Let* $\mathbf{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$, $\mathbf{SK} = (\mathbf{s}, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k, \mathbf{s}', \mathbf{k}', \mathbf{t}', ', s', h', k')$. *Suppose* s *and* s' *has been known by the adversary. Then,* h *and* h' *must be exhaustively searched in order to recover other secret values in* $\mathbf{SK}$ *from* $\mathbf{PK}$.

Proof. Let ith sample of $\mathbf{PK}$ be $(\mathbf{a}_i, b_i, \mathbf{a}'_i, b'_i)$. Since s and s' are assumed to be known by the adversary, we just let $s = 1$ and $s' = 1$ in this proof. Then, from the pubic key, the adversary obtains the following equations:

$$b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + r_i \bmod q \tag{13}$$

$$b'_i = \langle \mathbf{a}'_i, \mathbf{s}' \rangle + r'_i \bmod q \tag{14}$$

We know that $a\_max < p < h$ and the number of public key samples m is limited. Thus, in addition to the original values of $\mathbf{s}$ and $\mathbf{s}'$ specified in the private key, there can be other values of $\mathbf{s}$ and $\mathbf{s}'$, denoted by $\tilde{\mathbf{s}}$ and $\tilde{\mathbf{s}'}$, which satisfy the equations below, as confirmed by experiments in [8].

$$b_i = \langle \mathbf{a}_i, \tilde{\mathbf{s}} \rangle + \tilde{r}_i \bmod q \tag{15}$$

$$b'_i = \langle \mathbf{a}'_i, \tilde{\mathbf{s}'} \rangle + \tilde{r}'_i \bmod q \tag{16}$$

Thus, from each possible value of $\tilde{\mathbf{s}}$ and $\tilde{\mathbf{s}'}$, the adversary gets a pair of $\tilde{r}_i$ and $\tilde{r}'_i$. At this point, the adversary has to guess the moduli h and h' for the possibility of solving the equations below.

$$\tilde{r}_i = (\langle \mathbf{a}_i, \mathbf{k} \rangle + \langle \mathbf{a}'_i, \mathbf{t} \rangle + k * u_i \bmod p) + \langle \mathbf{a}_i, \mathbf{z} \rangle \bmod h \tag{17}$$

$$\tilde{r}'_i = (\langle \mathbf{a}'_i, \mathbf{k}' \rangle + \langle \mathbf{a}_i, \mathbf{t}' \rangle + k' * u_i \bmod p) + \langle \mathbf{a}'_i, \mathbf{z}' \rangle \bmod h' \tag{18}$$

□

Note that in the above proof, the correct guess of h and h' is just a necessary condition to solve equations (17) and (18), i.e., not a sufficient condition. The search space of h and h' is $\mathbf{Z}_p$

3.3 CPA-Security and CCA-Security of Compact-LWE-MQ$^{\mathrm{H}}$

Let $\mathbf{v} \in \mathbb{Z}_p^{n'}$ be the input to the encryption algorithm. It includes the plaintext in some of its entries, while other entries must include random numbers in order to be CPA-secure. Otherwise, according to Theorem 2, in the standard CPA indistinguishability experiment [5], the adversary simply guesses each message sent to the challenger, and the correct guess will return shorter solutions to the linear equations derived from the ciphertext and the public key, as shown in the proof of Theorem 2. If $\mathbf{v}$ includes a uniform random number from $\mathbb{Z}_p$, the probability of correctly guessing $\mathbf{v}$ is a negligible function with respect to the size of p in bits.

In practical applications, if $\mathbf{v}$ already contains uniformly sampled random values and is long enough, the encryption algorithm does not need to randomize $\mathbf{v}$ by adding extra random components. Theorem 3 ensures that the adversary cannot guess efficiently in the CPA indistinguishability experiment by recovering the secret component $\mathbf{s}$ and $\mathbf{s}'$.

Note that currently Compact-LWE-MQ$^{\mathrm{H}}$ is not CCA-secure. Let x be a small integer and $\mathbf{c}$ be the ciphertext encrypting one of two messages from the adversary in the standard CCA indistinguishability experiment [5]. Then, $\mathbf{c}$ and $\mathbf{c} * x$ decrypt into the

same message, because the decryption of $\mathbf{c} * x$ returns $\mathbf{g} * x$ (and hence $\mathbf{G} * x$) and $\mathbf{y} * x$, leading to

$$(\mathbf{G} * x)^{-1} * (\mathbf{y} * x) = \mathbf{G}^{-1} * \mathbf{y} \bmod p.$$

Thus, in the CCA experiment, the adversary can send $\mathbf{c} * x$ to the decryption oracle for decryption. In addition, there is another chosen ciphertext attack. Suppose $\mathbf{c}$ and $\mathbf{c}'$ encrypts the same message and $\mathbf{c}_a$ is a new ciphertext obtained from the component wise addition of $\mathbf{c}$ and $\mathbf{c}'$. Then, by sending the decryption oracle $\mathbf{c}_a$, the adversary can recover the message.

However, we can revise Compact-LWE-MQ$^{\mathtt{H}}$ slightly, as described below, to make it non-malleable in both attack cases.

- Add a random value $w \in \mathbb{Z}_p$ in the private key $\mathbf{SK}$ and also $\widehat{\mathbf{SK}}$.
- For u_i randomly selected in the generation of $\mathbf{PK}$, ensure that

$$\Sigma_{i=0}^{m-1}(\langle \mathbf{a'}_i, \mathbf{t} \rangle * k^{-1} + u_i + \langle \mathbf{a}_i, \mathbf{t'} \rangle * k'^{-1}) = w \bmod p.$$

- In Algorithm 2 of generating ciphertext component, change the initial value of $\mathbf{l}'$ (line 2) into

$$\mathbf{l}' = \mathbf{1} * (\Sigma_{i=0}^{n'-1} \mathbf{v}[i] \bmod p),$$

  where $\mathbf{1}$ is a m-dimensional vector containing integer 1 for each entry.
- In Algorithm 3 of decryption, change the code in line 10 into

$$\mathbf{g} = \mathbf{g} + (g + w).$$

We analyze the non-malleability of revised Compact-LWE-MQ$^{\mathtt{H}}$ briefly for the first attack case. With this updated Compact-LWE-MQ$^{\mathtt{H}}$, from $\mathbf{c} * x$, the decryption algorithm gets $\mathbf{g} * x + \mathbf{1} * w$, instead of $\mathbf{g} * x$ as above. Let $\mathbf{G}'$ be the matrix derived from $\mathbf{g} * x + \mathbf{1} * w$ in the decryption of $\mathbf{c} * x$ and $\mathbf{G}$ be the matrix derived from $\mathbf{g} + \mathbf{1} * w$ when decrypting $\mathbf{c}$. Then, we have

$$\mathbf{G'}^{-1} * (\mathbf{y} * x) \neq \mathbf{G}^{-1} * \mathbf{y} \bmod p.$$

4 Implementation and Attack Analysis

A prototype of Compact-LWE-MQ$^{\mathtt{H}}$ for CCA-security has been implemented with SageMath and is included in Appendix. In this section, we present a configuration of parameters, aimed at 128-bit security. Then, we use experiments to do security evaluation.

4.1 Configuration of Parameters

The configuration of parameters is shown in Table 1, where p has more than 128 bits, and q around 394 bits.

As stated in Theorem 3, both h and h' must be searched in the range of $\mathbf{Z}_p$ as a necessary step for performing attacks to the secret key. Since p is above 128 bits, only h or h' can guarantee the 128-bit security level for $\mathbf{SK}$. Hence, we do not need to quantify the extra guarantee from other secret components in $\mathbf{SK}$.

p	a_max	n	m	q
$2^{128}+51$	2^{56}	4	24	394 bits

Table 1: Configuration of Parameters or Parameter Size

In this configuration, the size of public key in bytes is close to

$$2 * m * (n * \log_2(a\_max) + \log_2(q))/8 = 2 * 24 * (4 * 56 + 394)/8 = 3708.$$

Suppose the plaintext to be encrypted is $v_1 \in \mathbb{Z}_p$. Note that v_1 might be known by the adversary with very high probability, just like in the CPA indistinguishability experiment. According to Theorem 2, v_1 must be padded with extra random numbers to make it hard to guess, before it is passed to the encryption algorithm.

Hence, the minimum of n' must be 2 in this configuration and the vector $\mathbf{v}$ passed to the encryption algorithm can be prepared as:

$$\mathbf{v}[0] \leftarrow \mathbb{Z}_p \texttt{ and } \mathbf{v}[1] = \mathbf{v}[0] \oplus v_1.$$

Similarly, if there are two plaintexts $v_1 \in \mathbb{Z}_p$ and $v_2 \in \mathbb{Z}_p$, which are not uniformly distributed, then we let $n' = 3$ and the vector $\mathbf{v}$ can be:

$$\mathbf{v}[0] \leftarrow \mathbb{Z}_p, \mathbf{v}[1] = \mathbf{v}[0] \oplus v_1 \texttt{ and } \mathbf{v}[2] = \mathbf{v}[0] \oplus v_2.$$

The size of ciphertexts depends on n'. When $n' = 2$, the size of the ciphertext in bytes is around $n' * 2 * (4 * (128 + 56 + \log_2(24)) + 394)/8 = 574$ bytes. The decryption failure for this configuration has the probability $\frac{1}{2^{128}+51}$.

4.2 Experiments

We use experiments to estimate whether m is big enough for the expected 128-bit security level, and conduct attacks to ciphertexts. The SageMath script for such attacks is also in Appendix.

4.2.1 Estimation of m If m is too small, there can be a small number of solutions for equations (1) and (2), and other equations (3) and (4), (5)and (6), (7) and (8) as well.

We have used experiments to evaluate whether $m = 24$ is big enough to allow more than 2^{128} solutions to those equations. The lattice developed in [2] is used to solve equations.

In equations (1) and (2), x_{10} (or other variables x_{1i} for $1 \leq i \leq m-1$) can be an integer between 0 and $p-1$. If for each $x_{10} \in \mathbb{Z}_p$ there is a different solution for other variables x_{1i} in equations (1) and (2), then there must be more than 2^{128} solutions to (1) and (2), because p is above 128 bits in the proposed configuration.

In our experiment, we select three sets of sample values for x_{10} to check the existence of distinct solutions for x_{1i} ($1 \leq i \leq m-1$). The first set is for x_{10} from 0 to 1023, the second set from $\lfloor p/2 \rfloor$ to $1023 + \lfloor p/2 \rfloor$, and the third set from $p - 1024$ to

$p-1$. In the test of all three sets, the distinct solutions exist for all other variables x_{1i} $(1 \leq i \leq m-1)$.

In addition, when m is as small as 8, with other parameters not changed, there exists only one solution to equations (1) and (2). Hence, $m = 24$ is big enough for 2^{128} distinct solutions for x_{1i}.

4.2.2 Attack Experiments In this experiment, we confirm that an arbitrary solution of x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} obtained by solving equations (1)-(8) cannot make equations (9) and (10) hold; that is, the solution of x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} cannot be used to recover $\mathbf{v}$.

In this experiment, we first get one solution for x'_{1i}, x_{2i}, and x'_{2i} by solving equations (3)-(8). Then, we generate different solutions for x_{1i} from equations (1) and (2). Our experiment confirmed that the solutions to equations (1)-(8) cannot satisfy overdetermined equations (9) and (10) for $0 \leq i \leq m-1$ and hence $\mathbf{v}$ cannot be recovered.

Another experiment is conducted to attack ciphertexts by exhaustively guessing $\mathbf{v}$ by choosing a small value for p, i.e. $p = 257$. With guessed $\mathbf{v}$, equations (1)-(10) become a linear system. In this experiment, we solve this system by guessing different values of $\mathbf{v}$, including the correct one.

Our experiment shows that x_{1i}, x'_{1i}, x_{2i}, and x'_{2i} have solution values roughly in the range of $\mathbb{Z}_p$, if $\mathbf{v}$ is correctly guessed; otherwise, there are no valid solutions for both versions of Compact-LWE-MQ$^{\mathtt{H}}$, or the solutions contain values much bigger than p in the CPA version of Compact-LWE-MQ$^{\mathtt{H}}$.

5 Related Works

Compact-LWE-MQ$^{\mathtt{H}}$ demonstrates the feasibility of defining public encryption without assuming hard computational problems. There is no existing work sharing the same purpose.

The public key samples in Compact-LWE-MQ$^{\mathtt{H}}$ is syntactically similar to LWE samples [9] at its top layer. However, Compact-LWE-MQ$^{\mathtt{H}}$ allows multiple solutions to secret vectors from the samples, while it is not the case for LWE. Thus, when n is a very small integer (e.g., $n = 4$) as configured for Compact-LWE-MQ$^{\mathtt{H}}$ in our evaluation, LWE samples can be solved to find the secret vector. More importantly, the ciphertexts in Compact-LWE-MQ$^{\mathtt{H}}$ can be represented as equations with plaintexts appeared in quadratic (MQ) terms, while the public encryption scheme in [9] (and other lattice-based schemes) treat plaintetxs as a linear term, which is vulnerable to lattice-based attacks for small parameters.

There are algorithms to solve MQ problems, such as [3, 4]. However, note that Compact-LWE-MQ$^{\mathtt{H}}$ is not dependent on the assumed hardness of MQ problems. One major step in all exiting algorithms solving MQ problems is to derive an initial system of linear equations, so that MQ problems can be solved by starting with the linear equations and then recursively finding the values of variables in quadratic terms. This step does not help attack Compact-LWE-MQ$^{\mathtt{H}}$, because its ciphertext is already a stratified system and its security is analyzed in Theorem 2. In addition, the ciphertext equations

in Compact-LWE-MQ$^{\mathtt{H}}$ are defined over two moduli (q and p), while equations in MQ problems are usually defined over one modulus for all equations.

The definition of public key Compact-LWE-MQ$^{\mathtt{H}}$ is partially influenced by Compact-LWE [8], in which ciphertexts are defined as a linear system. The ciphertexts in Compact-LWE have been attacked [2, 7]. However, all attacks to Compact-LWE are not applicable to Compact-LWE-MQ$^{\mathtt{H}}$, because ciphertexts in Compact-LWE-MQ$^{\mathtt{H}}$ is a stratified system of linear and quadratic equations.

6 Conclusion

In this paper, we have developed a public key encryption scheme Compact-LWE-MQ$^{\mathtt{H}}$, with the aim to demonstrate the feasibility of using hardness facts, rather than hardness assumptions, as a new principle of designing public key encryption. The factually hard problems ensure the plaintexts and secret values can only be recovered by brute-force search. Compact-LWE-MQ$^{\mathtt{H}}$ is simple to understand, facilitating the thorough analysis of its security from researchers with different backgrounds. If there is no attacks found in a short period of time, then there might be no attacks other than the brute-force one in any cases, reducing the uncertainty of public key encryption.

References

1. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing. pp. 99–108. STOC '96 (1996)
2. Bootle, J., Tibouchi, M., Xagawa, K.: Cryptanalysis of compact-lwe. In: Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10808, pp. 80–97. Springer (2018)
3. Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Preneel, B. (ed.) Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding. Lecture Notes in Computer Science, vol. 1807, pp. 392–407. Springer (2000)
4. Faugère, J.: A new efficient algorithm for computing gröbner bases (f4). Journal of Pure and Applied Algebra 139(1-3), 61–88 (1999)
5. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman & Hall/CRC (2014)
6. 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)
7. Li, H., Liu, R., Pan, Y., Xie, T.: Cryptanalysis of compact-lwe submitted to NIST PQC project. IACR Cryptol. ePrint Arch. 2018, 20 (2018)
8. Liu, D., Li, N., Kim, J., Nepal, S.: Compact-lwe: Enabling practically lightweight public key encryption for leveled iot device authentication. IACR Cryptol. ePrint Arch. 2017, 685 (2017), http://eprint.iacr.org/2017/685
9. 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)

10. Yasuda, T., Dahan, X., Huang, Y., Takagi, T., Sakurai, K.: MQ challenge: Hardness evaluation of solving multivariate quadratic problems. IACR Cryptol. ePrint Arch. (2015), http://eprint.iacr.org/2015/275

Appendix A Compact-LWE-MQ^{H} Implementation

```
# ================================
# Compact-LWE-MQ^{H}, CCA version
# ================================

set_random_seed(1)

p = next_prime(2^128)
n = 5
m = 24
a_max = 2^56

# ========================
def prikey_gen():
  h = next_prime(m*p*p+randint(0, p-1))
  hp = next_prime(m*p*p+randint(0, p-1))
  q = next_prime(m*p*(h+hp)+randint(0, p-1))

  S = vector(ZZ,[randint(0, q-1) for _ in range(n)])
  Sp = vector(ZZ,[randint(0, q-1) for _ in range(n)])

  K = vector(ZZ,[randint(0, p-1) for _ in range(n)])
  Kp = vector(ZZ,[randint(0, p-1) for _ in range(n)])

  T = vector(ZZ,[randint(0, p-1) for _ in range(n)])
  Tp = vector(ZZ,[randint(0, p-1) for _ in range(n)])

  Z = vector(ZZ,[randint(0, h-1) for _ in range(n)])
  Zp = vector(ZZ,[randint(0, hp-1) for _ in range(n)])

  k = randint(1, p-1)
  kp = randint(1, p-1)
  w = randint(0, p-1)

  s = randint(1, q-1)
  sp = randint(1, q-1)

  return S,Sp,s,sp,K,Kp,k,kp,T,Tp,Z,Zp,h,hp,w,q

def pubkey_gen(q,S,Sp,s,sp,K,Kp,k,kp,T,Tp,Z,Zp,h,hp,w):
  A1 = random_matrix(ZZ,m,n,x=0,y=1)
  A2 = random_matrix(ZZ,m,n,x=0,y=1) #A1
  b1 = vector(ZZ, [0 for _ in range(m)])
  b2 = vector(ZZ, [0 for _ in range(m)])

  kInv = inverse_mod(k, p)
  kpInv = inverse_mod(kp, p)
```

```
  for i in range(m):
    ui =  randint(0,p-1)

    A1[i,:] = vector(ZZ, [randint(0, a_max -1) for _ in range(n)])
    A2[i,:] = vector(ZZ, [randint(0, a_max -1) for _ in range(n)])

    if (i<m-1):
      w = (w - (A2[i]*T*kInv+ui+A1[i]*Tp*kpInv))%p
    else:
      ui = (w - (A2[i]*T*kInv+A1[i]*Tp*kpInv))%p

    r = ((K*A1[i]+T*A2[i]+k*ui)%p + Z*A1[i])%h
    rp = ((Kp*A2[i]+Tp*A1[i]+kp*ui)%p + Zp*A2[i])%hp
    e =  randint(0,p-1)
    ep = randint(0,p-1)
    b1[i] = (A1[i]*S +   s*(r) )%q
    b2[i] = (A2[i]*Sp + sp*(rp))%q

  return A1, b1, A2, b2

def encS(A1,b1,A2, b2, L, v, index, q):
  np = len(v)
  l1 =  L[index]
  l2 =  vector(ZZ, [sum(v)%p for i in range(m)])

  for i in range(np):
    l2 = l2 + L[(index+i)%np]*v[i]
    l2 =  l2%p

  ca1 = l1 * A1
  cb1 = (l1 * b1) % q
  ca2 = l2 * A2
  cb2 = (l2 * b2) % q

  c = (ca1, cb1, ca2, cb2)
  return c

def enc(q, A1,b1,A2, b2, v):
  np = len(v)
  L = random_matrix(ZZ,np,m,x=0,y=0)
  for i in range(np):
    l = vector(ZZ,[randint(0, p-1) for _ in range(m)])
    L[i,:] = l

  C=[]
  for i in range(np):
    if (i%2==0):
      c = encS(A1,b1,A2, b2, L, v, i, q)
    else:
```

```
      c = encS(A2,b2,A1, b1, L, v, i, q)

    C = C+ [c]

  return C


def decS(S,Sp,c,s,sp,K,Kp,q,k,kp,T,Tp,Z,Zp,h,hp):
  sInv = inverse_mod(s, q)
  spInv = inverse_mod(sp, q)
  kInv = inverse_mod(k, p)
  kpInv = inverse_mod(kp, p)

  ca1,cb1,ca2, cb2 = c
  d = sInv*(cb1-S*ca1)%q
  d  = (d-Z*ca1)%h
  g = (kInv*(d-K*ca1)+Tp*ca1*kpInv)%p

  dp = spInv*(cb2-Sp*ca2)%q
  dp = (dp-Zp*ca2)%hp
  y = (kpInv*(dp-Kp*ca2)+T*ca2*kInv)%p

  return g, y


def dec(q, S, Sp, C, s, sp, K, Kp, k, kp, T, Tp, Z, Zp, h, hp, w):
  y = []
  g = []
  np = len(C)
  for i in range(np):
    if (i%2==0):
      g1, y1= decS(S,Sp,C[i],s,sp,K,Kp,q,k,kp,T,Tp,Z,Zp,h,hp)
    else:
      g1, y1= decS(Sp,S,C[i],sp,s,Kp,K,q,kp,k,Tp,T,Zp,Z,hp,h)

    y = y + [y1]
    g = g + [(g1+w)%p]

  Rp = Integers(p)
  G = random_matrix(Rp,np,np)
  for i in range(np):
    G[i,:] = vector(Rp, g[i:np]+g[0:i])

  try:
    v = G.inverse()*vector(Rp, y)
  except:
    return -1

  return list(v)
```

```
#correctness
correct = 0
failure = 0
np = 2 #3, 5, 7, 9, 11, 13, ..., 101, ...

S, Sp, s, sp, K, Kp, k, kp, T, Tp, Z, Zp, h, hp, w, q = prikey_gen()
A1,b1,A2,b2 = pubkey_gen(q,S,Sp,s,sp,K,Kp,k,kp,T,Tp,Z,Zp,h,hp,w)

for i in range(10):
  v =[randint(0,p-1) for _ in range(np)]
  C = enc(q, A1,b1,A2, b2, v)
  dec_v = dec(q,S,Sp,C,s,sp,K,Kp,k,kp,T,Tp,Z,Zp,h,hp,w)
  if(v==dec_v):
    correct = correct+1

  if(dec_v==-1):
    failure = failure+1

print ("Correctness check: ", correct, failure)
```

Appendix B Ciphertext Attack

```
# ========================
# Ciphertext Attack Experiment
# with latex L from [2,7]
# ========================

def short_x(A1,b1,ca1,cb1, q):
  pa=q
  pa2=pa
  pa3=1

  L=block_matrix(ZZ, \
   [[1, 0,                      -pa*ca1.row(), -pa2 * cb1 ],\
   [0, pa3*identity_matrix(m), pa*A1,    pa2 * b1.column()],\
   [0, 0,                       0,                    pa2*q]
   ])

  L=L.LLL()

  #index of first non-zero entry in the first column of L
  idx=next((i for i,x in enumerate(L.column(0).list()) if x!=0))

  x = vector(ZZ,L[idx][1:(m)+1]/pa3) if L[idx][0] == 1 \
                        else vector(ZZ,-L[idx][1:(m)+1]/pa3)

  return x

def short_x1(A1,b1,ca1,cb1, x11, q):
  pa=q
  pa2=pa
  pa3=1

  ca1p = ca1 - x11*A1[0]
  cb1p = (cb1 - x11*b1[0])%q

  A10 = A1[0,:]
  b10 = b1[0]
  if(x11>0):
    A1[0,:] = vector(ZZ, [0 for _ in range(n)])
    b1[0] = 0

  L=block_matrix(ZZ, \
   [[1, 0,                      -pa*ca1p.row(),  -pa2 * cb1p], \
   [0, pa3*identity_matrix(m),  pa*A1,     pa2 * b1.column()], \
   [0, 0,                       0,                    pa2*q]
   ])
```

```
  L=L.LLL()

  #index of first non-zero entry in the first column of L
  idx=next((i for i,x in enumerate(L.column(0).list()) if x!=0))

  x = vector(ZZ,L[idx][1:(m)+1]/pa3) if L[idx][0] == 1 \
                          else vector(ZZ,-L[idx][1:(m)+1]/pa3)

  if(x11>0):
    x[0] = x11

  A1[0,:] = A10
  b1[0] = b10

  return x

S, Sp, s, sp, K, Kp, k, kp, T, Tp, Z, Zp, h, hp, w, q = prikey_gen()
A1,b1,A2,b2 = pubkey_gen(q,S,Sp,s,sp,K,Kp,k,kp,T,Tp,Z,Zp,h,hp,w)

v =[randint(0,p-1) for _ in range(2)]
C = enc(A1,b1,A2, b2, v)
ca1,cb1,ca2, cb2 = C[0]
ca3,cb3,ca4, cb4 = C[1]

x1p = short_x(A2,b2,ca2,cb2, qp)
print ("Check x1p:", x1p*A2==ca2, (x1p*b2)%qp == cb2)
x2 = short_x(A2,b2,ca3,cb3, qp)
print ("Check x2:", x2*A2==ca3, (x2*b2)%qp == cb3)
x2p = short_x(A1,b1,ca4,cb4, q)
print ("Check x2p:", x2p*A1==ca4, (x2p*b1)%q == cb4)

for x11 in range(2):
  x1 = short_x1(A1,b1,ca1,cb1, x11, q)
  print ("Check x1:", x1*A1==ca1, (x1*b1)%q == cb1)

  Rp = Integers(p)
  G = matrix(Rp, [[x1[0]+1, x2[0]+1], [x2[0]+1, x1[0]+1]])
  vp = G.inverse()*vector(Rp, [x1p[0], x2p[0]])

  consist = 0
  for j in range(m):
    if ((x1[j]*vp[0]+x2[j]*vp[1]+vp[0]+vp[1])%p==(x1p[j])%p and \
       (x1[j]*vp[1]+x2[j]*vp[0]+vp[0]+vp[1])%p==(x2p[j])%p):
       consist= consist + 1

  if(consist==m):
    print ("+++++successful attack++++")
  else:
    print ("----failed attack----", consist)
```