

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^H¹ to demonstrate the feasibility of constructing 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 propose in this work are stratified system of linear and quadratic equations, and layered learning with relatively big errors. The factually hard problems are characterized by their layered structures, which ensure that the secrets at a lower layer can only be recovered after the secrets in a upper layer have been found *correctly* (i.e., leading to consistent lower layer secrets, not necessarily the original upper layer ones). On the other hand, without knowing the secrets in the lower layer, the upper layer sub-problem can only be solved by exhaustive search.

Based on the structure of factually hard problems, we prove that without brute-force search the adversary cannot recover plaintexts or correct private key, and then discuss CPA-security and CCA-security of Compact-LWE-MQ^H. We have implemented Compact-LWE-MQ^H with a number of lines of SageMath code. Simplicity of Compact-LWE-MQ^H makes it easy for understanding, cryptanalysis, and implementation.

In our configuration for 128-bit security, the dimensional parameter is $n = 4$ (n has the same meaning as in LWE). For such a tiny parameter, the current analysis tools like LLL lattice reduction algorithm are already efficient enough to perform attacks if the security claim of Compact-LWE-MQ^H does not hold. That is, the security of Compact-LWE-MQ^H is not assumed with the capability of cryptanalysis tools. SageMath code of verifying Compact-LWE-MQ^H security is also included in Appendix.

1 Introduction

Public key encryption is usually constructed around various assumptions of computationally hard problems as its security foundation. The well-known hardness assumptions include the problems of factoring the product of big prime numbers, computing

* Corresponding author: dongxi.liu@csiro.au

¹ The scheme is the same as in the previous version of the report (10 Aug 2020); only explanations and discussions are changed.

discrete logarithms over finite fields, multivariate quadratic problems [9], 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 a countermeasure. On the other hand, if a new computing paradigm (e.g., quantum computing) makes hardness assumptions no longer hold, new public key encryption schemes have to be designed, as being done by NIST Post-Quantum Cryptography Standardization project.

In this paper, we demonstrate the feasibility of designing public key encryption based on hardness facts, rather than assumptions. To this end, we design the public key encryption scheme Compact-LWE-MQ^H, which has its security based on the following factually hard problems.

1.1 Factual Hardness Problems

The factually hard problems described below have layered structures, with secrets contained in each layer. Any attempt to find secret values in a lower layer must be preceded by necessary operations to *correctly* find secret values in a upper layer. The correct condition means that the candidate secret values from the upper layer must lead to consistent secret values in a lower layer.

Without knowing the secrets in the lower layer, the upper layer subproblem can only be solved by exhaustively search in factually hard problems. Even configured with very small parameters, a factually hard problem still needs to be solved with brute-force search.

To define the problems below, we let h , h' , and p be three prime numbers, and m , n , a_max three integers. We require $q > m * p * (h' + h)$, $h > m * p * p$, $h' > m * p * p$, $m > 2 * n$, and $p > a_max$.

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

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

where \mathbf{a}_i and \mathbf{a}'_i are uniformly sampled from $\mathbb{Z}_{a_max}^n$, and $r_i \in \mathbb{Z}_h$ and $r'_i \in \mathbb{Z}_{h'}$ are defined as,

$$r_i = u_i + \langle \mathbf{z}, \mathbf{a}_i \rangle \bmod h, r'_i = u_i + \langle \mathbf{z}', \mathbf{a}'_i \rangle \bmod h',$$

where u_i is uniformly sampled from \mathbb{Z}_p . In the samples, \mathbf{a}_i , \mathbf{a}'_i , b_i , and b'_i are public values. Finding the candidate values of \mathbf{z} and \mathbf{z}' must be preceded by the operations to find the values of \mathbf{s} , \mathbf{s}' , h and h' .

Compared with Learning with Errors (LWE) [8], 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. Intuitively, since $\mathbf{a}_i[0]$ is small, $r_i + \mathbf{a}_i[0]$ can be a legitimate error term (i.e., its value is in \mathbb{Z}_h) for $0 \leq i \leq m - 1$ with a high chance; thus $\tilde{\mathbf{s}}$ is a valid solution, where $\tilde{\mathbf{s}}[0] = \mathbf{s}[0] - 1$ and $\tilde{\mathbf{s}}[j] = \mathbf{s}[j]$ for $1 \leq j \leq n - 1$.

Correct values of \mathbf{z} and \mathbf{z}' are required to keep u_i the same in r_i and r'_i in each sample. Note that we do not require the secret values must be the same as the original ones for correctness. To find a candidate value for \mathbf{z} and \mathbf{z}' , the adversary must first have a solution to \mathbf{s} and \mathbf{s}' , which leads to some values for r_i and r'_i (not necessarily the original ones). And then, h and h' have to be guessed to proceed. The experiment code in Appendix confirms that an arbitrary guess of h and h' cannot fulfill the correctness requirement of \mathbf{z} and \mathbf{z}' ; so exhaustive search of moduli h and h' is necessary.

Even worse for the adversary, there might be no h and h' that can generate correct \mathbf{z} and \mathbf{z}' for some solutions to \mathbf{s} and \mathbf{s}' . For instance, this is the case, where \mathbf{s} is $\tilde{\mathbf{s}}$ as defined above and \mathbf{s}' is the original one.

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 cb , cb' , b_i , and b'_i are public values,

$$\begin{aligned} cb &= \sum_{i=0}^{m-1} x_i * b_i \bmod q \\ cb' &= \sum_{i=0}^{m-1} x'_i * b'_i \bmod q \end{aligned}$$

and m multivariate quadratic (MQ) equations,

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

In this problem, x_i and x'_i ($0 \leq i \leq m - 1$) are secret values in the upper layer, and v is the secret value in the lower layer.

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

We call the system with above equations stratified, because the values of x_i and x'_i (or at least one of them) can only be obtained by solving the corresponding linear equations, and then these values are used to check whether a correct v can be determined from the quadratic equations. 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 solution space for 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 in the solution space has to be found and checked. In Compact-LWE-MQ^H, v is generalized into a vector.

1.2 Overview of Compact-LWE-MQ^H

Compact-LWE-MQ^H is constructed around the above hardness facts, with further security enhancement. Briefly, the first hardness fact ensures a correct 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 Compact-LWE-MQ^H, the private key includes a number of extra secret components, in addition to those listed in the first factually hard problem. First, another lower layer of samples are added into r_i and $r_{i'}$, as to be detailed in the construction later. Second, r_i and $r_{i'}$ are multiplied with secret values in \mathbb{Z}_q before they are added with the corresponding inner products.

The plaintext in Compact-LWE-MQ^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^H achieves CPA-security when \mathbf{v} is random enough for the adversary. After being slightly revised, Compact-LWE-MQ^H is also CCA-secure.

In the security proof of Compact-LWE-MQ^H, we identify the necessary operations (or steps) that have to be taken by the adversary to manipulate the public keys and ciphertexts. The factually hard problems underlying Compact-LWE-MQ^H ensure that the adversary has to do brute-force search at these necessary steps, enforcing by their layered structure.

The construction of Compact-LWE-MQ^H is *conceptually* simple. The following concepts used in existing post-quantum public key encryption algorithms are not needed to understand Compact-LWE-MQ^H: polynomials, complex probability distributions or sampling techniques, complex coding/decoding, hashing. The basic knowledge of modular arithmetic, matrix and vector operations (i.e., inner product, inverse of matrix, multiplication of matrix and vector), and uniform distribution is sufficient to understand and analyze Compact-LWE-MQ^H for a large cohort of people. The selection of parameters is also straightforward for a desired security level.

Compact-LWE-MQ^H is straightforward to implement due to its simple construction. We have implemented Compact-LWE-MQ^H in SageMath and the implementation is in Appendix. In a configuration for 128-bit security, we have parameters $m = 24$ and $n = 4$ (with the same meaning as in LWE), and p with 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^H in this paper.

Since m and n take tiny values, the existing cryptanalysis tools like LLL algorithm are already efficient enough to completely attack Compact-LWE-MQ^H in a short period of time if Compact-LWE-MQ^H has a structure flaw. In addition, any suspicion to the security of Compact-LWE-MQ^H can also be checked with the existing cryptanalysis tools. The SageMath code of verifying the security claims relevant to both factually hard problems is included in Appendix.

1.3 Notations

Given a positive integer q , \mathbb{Z}_q refers to the set $\{0, \dots, q-1\}$. 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 i th element of list \mathbf{a} is written as $\mathbf{a}[i]$ and i starts from 0. The sublist of list \mathbf{a} from its i th 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 $\mathbf{a} \parallel \mathbf{b}$. 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 $\text{len}(\mathbf{a})$. The prime number just after an integer n is denoted by $\text{np}(n)$. $\mathbf{0}$ means a zero vector, with its dimension derived from its usage context.

2 Construction of Compact-LWE-MQ^H

As a public-key encryption scheme, Compact-LWE-MQ^H is defined with three algorithms: key generation, encryption, and decryption. Compact-LWE-MQ^H partially shares its name with Compact-LWE [7], which has been broken. However, they are completely different on each algorithm in the schemes. Particularly, Compact-LWE (and other hardness assumption) does not have layered structure in their problems. The only similarity with Compact-LWE is that a noise in a sample can be bigger than elements in \mathbf{a}_i .

Compact-LWE-MQ^H is parameterized with four integers: n, m, p , and a_{max} . We require p be a prime, $a_{max} < p$, $2 * n < m < p$. 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 $\text{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$. 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 $(\mathbf{s}, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k, s', \mathbf{k}', \mathbf{t}', \mathbf{z}', s', h', k')$, generated in the steps below.

- $h = \text{np}(m * p * p + r)$, where $r \leftarrow \mathbb{Z}_p$.
- $h' = \text{np}(m * p * p + r')$, where $r' \leftarrow \mathbb{Z}_p$.
- $q = m * p * (h + h') + q'$, where $q' \leftarrow \mathbb{Z}_p$.
- $\mathbf{s} \leftarrow \mathbb{Z}_q^n$, $\mathbf{k} \leftarrow \mathbb{Z}_p^n$, $\mathbf{t} \leftarrow \mathbb{Z}_p^n$, $s' \leftarrow \mathbb{Z}_q^n$, $\mathbf{k}' \leftarrow \mathbb{Z}_p^n$, 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 and it is randomized with a number sampled from \mathbb{Z}_p ; additionally, we require $n * \log_2(a_max) + \log_2(p * p) > \log_2(q)$. All secret values in \mathbf{SK} and random values used below are randomly sampled from uniform distributions. Given \mathbf{SK} , we write $\widehat{\mathbf{SK}}$ for $(s', \mathbf{k}', \mathbf{t}', \mathbf{z}', s', h', k', \mathbf{s}, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k)$.

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 i th 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; so it has a layered structure. 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}' . Other secrets \mathbf{k} , \mathbf{k}' , \mathbf{t} , \mathbf{t}' , k , k' are hidden even deeper in this structure.

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^H is an element in $\mathbb{Z}_p^{n'}$, where n' is an odd integer when 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^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 invoked n' times; at the i th time ($0 \leq i \leq n' - 1$), the i th 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

Algorithm 1: enc for encryption

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

- 1 Let \mathbf{L} be a list with n' entries.
- 2 **for** $i = 0$ **to** $n' - 1$ **do**
- 3 | $\mathbf{L}[i] \leftarrow \mathbb{Z}_p^m$
- 4 **end**
- 5 $\mathbf{c} = []$
- 6 **for** $i = 0$ **to** $n' - 1$ **do**
- 7 | **if** $i \bmod 2 = 0$ **then**
- 8 | | $c = \text{encS}(\mathbf{PK}, \mathbf{L}, \mathbf{v}, i)$
- 9 | **end**
- 10 | **else**
- 11 | | $c = \text{encS}(\widehat{\mathbf{PK}}, \mathbf{L}, \mathbf{v}, i)$
- 12 | **end**
- 13 | $\mathbf{c} = \mathbf{c} + c$
- 14 **end**
- 15 **return** \mathbf{c}

Algorithm 2: encS for encrypting ciphertext component

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

- 1 $\mathbf{l} = \mathbf{L}[i]$
- 2 $\mathbf{l}' = \mathbf{0}$
- 3 **for** $j = 0$ **to** $n' - 1$ **do**
- 4 | $\mathbf{l}' = \mathbf{l}' + \mathbf{v}[j] * \mathbf{L}[(i + j) \bmod n'] \bmod p$
- 5 **end**
- 6 $\mathbf{ca}_1 = \mathbf{0}, cb_1 = 0, \mathbf{ca}_2 = \mathbf{0}, cb_2 = 0$
- 7 **for** $j = 0$ **to** $m - 1$ **do**
- 8 | Let $(\mathbf{a}_1, b_1, \mathbf{a}_2, b_2) = \mathbf{PK}'[j]$.
- 9 | $\mathbf{ca}_1 = \mathbf{ca}_1 + \mathbf{l}[j] * \mathbf{a}_1$
- 10 | $cb_1 = cb_1 + \mathbf{l}[j] * b_1 \bmod q$
- 11 | $\mathbf{ca}_2 = \mathbf{ca}_2 + \mathbf{l}'[j] * \mathbf{a}_2$
- 12 | $cb_2 = cb_2 + \mathbf{l}'[j] * b_2 \bmod q$
- 13 **end**
- 14 $c = (\mathbf{ca}_1, cb_1, \mathbf{ca}_2, cb_2)$
- 15 **return** c

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 `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: dec for decryption

```

input :  $\mathbf{SK}, \mathbf{c}$ 
output:  $\mathbf{v}$  or -1
1  $n' = \text{len}(\mathbf{c})$ 
2  $\mathbf{g} = [], \mathbf{y} = []$ 
3 for  $i = 0$  to  $n' - 1$  do
4   if  $i \bmod 2 = 0$  then
5      $g, y = \text{decS}(\mathbf{SK}, \mathbf{c}[i])$ 
6   end
7   else
8      $g, y = \text{decS}(\widehat{\mathbf{SK}}, \mathbf{c}[i])$ 
9   end
10   $\mathbf{g} = \mathbf{g} + g$ 
11   $\mathbf{y} = \mathbf{y} + y$ 
12 end
13 Let  $\mathbf{G}$  be a  $n' * n'$  matrix.
14 for  $i = 0$  to  $n' - 1$  do
15    $\mathbf{G}[i] = \mathbf{g}[i : n'] \parallel \mathbf{g}[0 : i]$ 
16 end
17 if  $\mathbf{G}$  is singular with respect to  $p$  then
18   return -1
19 end
20  $\mathbf{v} = \mathbf{G}^{-1} * \mathbf{y} \bmod p$ 
21 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 (e.g., by re-sampling some random elements or extending it to include more random elements).

2.3 Decryption

The decryption algorithm `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 $\text{dec}(\mathbf{SK}, \mathbf{c}) = \mathbf{v}$; 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 dec algorithm invokes the algorithm **decS**, defined in Algorithm 4, to return a pair of integers g and y in \mathbb{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 **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 **dec** algorithm derives the $n' \times n'$ 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: **decS** for decrypting ciphertext component

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

- 1 Let $\mathbf{SK}' = (s_1, \mathbf{k}_1, \mathbf{t}_1, \mathbf{z}_1, s_1, h_1, k_1, s_2, \mathbf{k}_2, \mathbf{t}_2, \mathbf{z}_2, s_2, h_2, k_2)$.
- 2 Let $s_1^{-1} \in \mathbb{Z}_{q_1}$, such that $s_1^{-1} * s_1 = 1 \bmod q$.
- 3 Let $s_2^{-1} \in \mathbb{Z}_{q_2}$, such that $s_2^{-1} * s_2 = 1 \bmod q$.
- 4 Let $k_1^{-1} \in \mathbb{Z}_p$, such that $k_1^{-1} * k_1 = 1 \bmod p$.
- 5 Let $k_2^{-1} \in \mathbb{Z}_p$, such that $k_2^{-1} * k_2 = 1 \bmod p$.
- 6 $d = s_1^{-1} * (cb_1 - \langle \mathbf{ca}_1, \mathbf{s}_1 \rangle) \bmod q$
- 7 $d = (d - \langle \mathbf{ca}_1, \mathbf{z}_1 \rangle) \bmod h_1$
- 8 $g = k_1^{-1} * (d - \langle \mathbf{ca}_1, \mathbf{k}_1 \rangle) + \langle \mathbf{ca}_1, \mathbf{t}_2 * k_2^{-1} \rangle \bmod p$
- 9 $d' = s_2^{-1} * (cb_2 - \langle \mathbf{ca}_2, \mathbf{s}_2 \rangle) \bmod q$
- 10 $d' = (d' - \langle \mathbf{ca}_2, \mathbf{z}_2 \rangle) \bmod h_2$
- 11 $y = k_2^{-1} * (d - \langle \mathbf{ca}_2, \mathbf{k}_2 \rangle) + \langle \mathbf{ca}_2, \mathbf{t}_1 * k_1^{-1} \rangle \bmod p$
- 12 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^H is deterministically correct if the ciphertext can be decrypted. A ciphertext 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 $\text{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$, $\mathbf{v} \in \mathbb{Z}_p^{n'}$ and $\mathbf{c} = \text{enc}(\mathbf{PK}, \mathbf{v})$. Then, $\text{dec}(\mathbf{SK}, \mathbf{c}) = \mathbf{v}$ with probability $\frac{p-1}{p}$, or $\text{dec}(\mathbf{SK}, \mathbf{c}) = -1$ 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

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

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

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

Still in Algorithm 4, from $d_1, d'_1, d_2,$ and d'_2 , the following values can be recovered.

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

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. \square

3 Security Analysis

The security of Compact-LWE-MQ^H is not based on computational hardness assumptions, so the conventional 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. The necessary steps are enforced by the layered structures in our factually hard problems. Theorem 2 and Theorem 3 below will show that brute-force search is necessary in the identified necessary steps.

Note that the security claims below (Theorem 2 and Theorem 3) do not rely on the size of parameters (n , m , p , and a_{max}); small size of parameters just means a small space for brute-force search. Hence, by choosing small parameters, existing cryptanalysis tools can be used to verify these security claims. On the contrary, for existing schemes based on hardness assumption, their security cannot be verified when parameters are small, because at this case they can be solved without brute-force search (e.g., factoring the product of two small primes).

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 $\text{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$, and $\mathbf{c} = \text{enc}(\mathbf{PK}, \mathbf{v})$, where $\mathbf{v} \in \mathbb{Z}_p^{n'}$. Then, from \mathbf{PK} and \mathbf{c} , the plaintext \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 public key \mathbf{PK} , the adversary can obtain a stratified system, consisting of the following linear equations,

$$\mathbf{ca}_1 = \sum_{i=0}^{m-1} x_{1i} * \mathbf{a}_i \quad (1)$$

$$cb_1 = \sum_{i=0}^{m-1} x_{1i} * b_i \text{ mod } q \quad (2)$$

$$\mathbf{ca}'_1 = \sum_{i=0}^{m-1} x'_{1i} * \mathbf{a}'_i \quad (3)$$

$$cb'_1 = \sum_{i=0}^{m-1} x'_{1i} * b'_i \text{ mod } q \quad (4)$$

$$\mathbf{ca}_2 = \sum_{i=0}^{m-1} x_{2i} * \mathbf{a}'_i \quad (5)$$

$$cb_2 = \sum_{i=0}^{m-1} x_{2i} * b'_i \text{ mod } q \quad (6)$$

$$\mathbf{ca}'_2 = \sum_{i=0}^{m-1} x'_{2i} * \mathbf{a}_i \quad (7)$$

$$cb'_2 = \sum_{i=0}^{m-1} x'_{2i} * b_i \text{ mod } q \quad (8)$$

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

$$x'_{1i} = \mathbf{v}[0] * x_{1i} + \mathbf{v}[1] * x_{2i} \text{ mod } p \quad (9)$$

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

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 $\sum_{i=0}^{m-1} x'_{1i} * \mathbf{a}'_i = \mathbf{ca}'_1$ and $\sum_{i=0}^{m-1} x_{2i} * \mathbf{a}'_i = \mathbf{ca}_2$ (or $\sum_{i=0}^{m-1} x'_{2i} * \mathbf{a}_i = \mathbf{ca}'_2$ and $\sum_{i=0}^{m-1} x_{1i} * \mathbf{a}_i = \mathbf{ca}_1$). Thus, equations (9) and (10) are reduced to:

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

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

In the equations (11) and (12), the values of $\sum_{i=0}^{m-1} x_{1i} * \mathbf{a}'_i$ and $\sum_{i=0}^{m-1} x_{2i} * \mathbf{a}_i$ are not known. Hence, in order 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. \square

With the implementation of Compact-LWE-MQ^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 $\mathbf{v}[1]$.

3.2 Attack to Public Keys

The aim of the adversary in this attack is to find a *correct* private key \mathbf{SK} from the public key \mathbf{PK} . Note that a correct \mathbf{SK} is not necessarily the original one; instead, it must guarantee the consistency of u_i in all samples.

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 possible attacks to recover candidate solutions to \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 $\sum_{i=1}^m (\mathbf{l}[i] * \mathbf{a}_i) = \mathbf{0}$ (and/or $\sum_{i=1}^m (\mathbf{l}[i] * \mathbf{a}'_i) = \mathbf{0}$). Then, the adversary can get $\sum_{i=1}^m (\mathbf{l}[i] * b_i) = s * \sum_{i=1}^m (\mathbf{l}[i] * r_i) \bmod q$. Then, the adversary can guess $\sum_{i=1}^m (\mathbf{l}[i] * r_i)$, which is much smaller than s .

In the parameter configuration proposed later, the size of p is 128 bits. Thus, guessing $\sum_{i=1}^m (\mathbf{1}[i] * r_i)$ still needs to search in a big space. However, we will assume the adversary has somehow obtained the original s and s' . We then focus on proving the necessity of exhaustively searching h and h' , two moduli in the bottom layer.

Theorem 3. *Let $\text{gen}(pp) = (\mathbf{SK}, \mathbf{PK})$, $\mathbf{SK} = (s, \mathbf{k}, \mathbf{t}, \mathbf{z}, s, h, k, s', \mathbf{k}', \mathbf{t}', \mathbf{z}', s', h', k')$. Suppose s and s' has been known by the adversary. Then, h and h' must be exhaustively searched to check whether secret values (i.e., $\mathbf{k}, \mathbf{k}', \mathbf{t}, \mathbf{t}', \mathbf{z}, \mathbf{z}', k$, and k') can lead to consistent u_i in each sample of \mathbf{PK} .*

Proof. Let i th 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 public key, the adversary obtains the following equations:

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

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

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 [7].

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

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

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 and checking the correctness of $\mathbf{k}, \mathbf{k}', \mathbf{t}, \mathbf{t}', \mathbf{z}, \mathbf{z}', k$, and k' , in terms that u_i in each sample is the same.

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

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

Moreover, an arbitrary guess of h and h' will not lead to correct secret values in the above equations, with the reason discussed below. Hence, an exhaustive search to h and h' is necessary.

Suppose the adversary has guessed h' and gets u_i from equations (18). Then, the adversary moves to solve equations (17), where u_i becomes known values. With different guesses of h , the expression $\langle \mathbf{a}_i, \mathbf{k} \rangle + \langle \mathbf{a}'_i, \mathbf{t} \rangle + k * u_i \bmod p$ in equations (17) is constrained to different values. The equation system consisting of such expressions and their values is over-determined because $2 * n < m$. An arbitrary guess of h cannot make this over-determined system have consistent solutions to \mathbf{k}, \mathbf{t} , and k . This is confirmed with an experiment in Appendix.

□

Note that for an arbitrary solution to $\tilde{\mathbf{s}}$ and $\tilde{\mathbf{s}}'$, there might not exist h and h' that can lead to correct secret values at the bottom layer; this point will be exploited in the future work to make Compact-LWE-MQ^H lightweight.

The search space of h and h' is \mathbf{Z}_p as defined by the key generation algorithm. Based on this theorem, the selection of parameters for a desired security level will be very straightforward.

3.3 CPA-Security and CCA-Security of Compact-LWE-MQ^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 in one of its entries \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 efficiently recovering a correct private key (not necessarily the original one).

Note that currently Compact-LWE-MQ^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 \mathbf{c}_a to the decryption oracle, the adversary can recover the message.

However, we can revise Compact-LWE-MQ^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

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

- In Algorithm 2 of generating ciphertext component, change the initial value of l' (line 2) into

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

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^H briefly for the first attack case. With this updated Compact-LWE-MQ^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

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

4 Implementation and Attack Analysis

A prototype of Compact-LWE-MQ^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. As to be seen below, for a desired security level, the parameters can be easily determined, without relying on the capability of any cryptanalysis algorithms like a lattice-reduction algorithm.

4.1 Configuration of Parameters

The configuration of parameters is shown in Table 1, where p has 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 find a correct private key. Since p is 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 $v_1 \in \mathbf{Z}_p$. 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:

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

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:

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

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 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, an arbitrary 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^H, or the solutions contain values much bigger than p in the CPA version of Compact-LWE-MQ^H.

5 Related Works

Compact-LWE-MQ^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^H is syntactically similar to LWE samples [8] at its top layer. However, Compact-LWE-MQ^H allows multiple solutions to secret vectors at the top layer 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^H in our evaluation, LWE samples can be solved to find the secret vector. More importantly, the ciphertexts in Compact-LWE-MQ^H can be represented as equations with plaintexts appeared in quadratic (MQ) terms, while the public encryption scheme in [8] (and other lattice-based schemes) treat plaintexts 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]. One major step in all existing 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^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^H are defined over two moduli at different layers (q and p), while equations in MQ problems are usually defined over one modulus for all equations (i.e., no layered structures in current MQ problems with all secrets at the same layer).

Compact-LWE-MQ^H shares its name with Compact-LWE [7], which have been attacked in [2, 6] by recovering plaintexts from ciphertexts. This is because ciphertexts are defined as a linear system in Compact-LWE; hence, with small dimension parameters, the current lattice-reduction tools can attack Compact-LWE efficiently. All attacks to Compact-LWE are not applicable to Compact-LWE-MQ^H, because ciphertexts in Compact-LWE-MQ^H is a stratified system of linear and quadratic equations and the upper layer subproblem has to be solved by exhaustive search when the lower layer secret is not known.

6 Conclusion

In this paper, we have developed a public key encryption scheme Compact-LWE-MQ^H, with the aim to demonstrate the feasibility of using hardness facts, rather than hardness

assumptions, as a new principle of constructing public key encryption. The factually hard problems ensure the plaintexts and secret values can only be recovered by taking brute-force search as one necessary step. Compact-LWE-MQ^H is simple to understand, facilitating the thorough analysis of its security from researchers with different backgrounds. With very small parameter configuration, the existing cryptanalysis tools are already efficient enough to attack Compact-LWE-MQ^H if its security claim does not hold. In other words, the security of Compact-LWE-MQ^H does not assume the capability of cryptanalysis tools or hardness of problems.

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. Li, H., Liu, R., Pan, Y., Xie, T.: Cryptanalysis of compact-lwe submitted to NIST PQC project. *IACR Cryptol. ePrint Arch.* 2018, 20 (2018)
7. 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>
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)
9. 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-MQH, CCA version
# =====

set_random_seed(1)

p = next_prime(2128)
n = 5
m = 24
a_max = 256

# =====
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)

```


Appendix C Check Arbitrary Guess of h

```
A1 = random_matrix(ZZ,3,3,x=0,y=7)
A2 = random_matrix(ZZ,3,3,x=0,y=7)

h = 23

Rp = Integers(h)
K = vector(ZZ,[randint(0, h) for _ in range(3)])

A1a = A1.change_ring(Rp)
A2a = A2.change_ring(Rp)

y1 = (A1a*K).lift()
y2 = (A2a*K).lift()

K1a = A1a.inverse()*y1
K2a = A2a.inverse()*y2

print ("---consistent solution with the correct modulus----")
print (K1a)
print (K2a)

h = 31 #19, 23, 29, 31, 37, 91, 1019

Rpp = Integers(h)

A1b = A1.change_ring(Rpp)
A2b = A2.change_ring(Rpp)
K1b = A1b.inverse()*y1
K2b = A2b.inverse()*y2

print ("----inconsistent solution with an arbitrary modulus")
print (K1b)
print (K2b)
```