

# 后量子密码学课程论文



**题目：从量子计算到后量子算法**

Shandong University

2023 年 4 月 26 日

## 摘 要

本文为后量子密码学课程论文，首先介绍了量子计算，并通过 Shor 算法展现了量子计算在密码学计算方面的强大能力，由此引出了接下来介绍的后量子密码算法。

在量子计算部分，首先对量子计算的主要概念和实现目标进行了概括，也就是通过量子电路对量子比特进行操作从而使其能测量时高概率塌缩到目标状态。主要以 Shor 算法为例，补充介绍了课程的 Shor 算法内容。通过一个计算实例来展现了量子算法如何同时对多个状态进行运算，直观展示了量子傅里叶变换在相位估计中的作用。此外，还通过微软的 Qiskit 平台对 Shor 算法进行了实现，进一步从电路层面理解了量子算法。

为了抗量子计算，需要构建后量子密码算法。本文以目前无有效量子求解算法的 LWE 问题入手，给出了 LWE 问题的概念和不同表示。并介绍了 LWE 问题构建的 Regev 公钥加密算法，从正确性、安全性等方面给出了分析，另外通过 python 成功实现了 Regev 公钥加密算法，加深了对算法的理解。最后进一步扩展了 LWE 问题在构建全同态加密算法中的作用。

**关键词:** 量子计算 后量子密码算法 Shor 算法 LWE 问题 Regev 加密算法

# 目 录

<b>1</b>	<b>量子计算及 Shor 算法</b>	<b>3</b>
1.1	量子计算 . . . . .	3
1.2	因子分解问题 . . . . .	4
1.3	量子傅里叶变换 . . . . .	5
1.4	Shor 算法 . . . . .	7
1.5	算法实例分析及实现 . . . . .	9
1.6	Shor 算法的改进 . . . . .	10
<b>2</b>	<b>后量子密码学</b>	<b>11</b>
2.1	LWE 问题 . . . . .	11
2.2	Regev 公钥加密算法 . . . . .	12
2.2.1	Regev 加密的正确性 . . . . .	12
2.2.2	Regev 加密的安全性 . . . . .	13
2.2.3	Regev 加密的实现 . . . . .	14
2.3	LWE 与全同态加密 . . . . .	14
	<b>参考文献</b>	<b>14</b>
	<b>附录 A Regev 公钥加密算法</b>	<b>15</b>
	<b>附录 B Qiskit Shor algorithm</b>	<b>17</b>

# 1 量子计算及 Shor 算法

## 1.1 量子计算

经典计算的基本单位是比特，一般表示为 0 和 1 的孤立状态。而量子计算的基本单位是量子比特 (quantum bit, 简记为 qubit)，一般处于叠加状态。对于量子比特，一般使用 Dirac notation，也就是记为  $|0\rangle$  和  $|1\rangle$ 。

当然量子比特和传统比特之间的区别是量子比特可以处在一种叠加态，也就是 0 态和 1 态的一种线性组合 [1]

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

一般地，单个量子比特处在叠加态，如果使用计算基向量进行测量则以一定的概率会塌缩到某一个基态。如果对  $|\phi\rangle$  这个量子比特进行测量，就会以  $|\alpha|^2$  塌缩到基态  $|0\rangle$  或者以  $|\beta|^2$  概率塌缩到基态  $|1\rangle$ 。当然在多量子比特的情况下，对某一个比特进行测量剩余比特的系数会按照比例分配，这里不再赘述。

对于经典比特，可以使用门电路对其运算，类似地，量子比特也可以通过量子门电路对量子比特进行线性变换。对量子比特进行运算是对于一个叠加状态进行运算，也就是可以看作同时对多个传统比特进行运算，但是最后测量时会以一定概率塌缩到某一个状态，构造量子算法就是使测量时能够以高概率塌缩到目标状态。

下面给出了一些常用的通用量子门，接下来将介绍量子算法在密码学领域的强大应用。













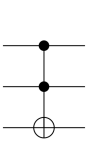
Operator	Gate(s)	Matrix
Pauli-X (X)	 	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP	 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

图 1: Quantum logic Gates

## 1.2 因子分解问题

RSA 密码体制的安全性依赖于大整数分解困难问题，对于大整数分解问题，可以把它转化为 Order Finding 问题，也就是元素求阶问题上。

首先假设要分解的大数为  $N$ ，如果  $N$  是偶数，自然有因子 2，问题容易解决；如果  $N$  不是偶数，那么可以从 1 到  $N-1$  之间选择一个和它互质的数  $a$ ，求  $a$  对  $N$  的阶  $r$ 。如果  $r$  是一个偶数，那么有

$$a^r = (a^{r/2})^2 = 1 \pmod{N} \quad (1)$$

因此，有  $(a^{r/2} - 1)(a^{r/2} + 1) = 0 \pmod{N}$ ，这样也就是这两个数至少

有一个含有  $N$  的公约数，可以使用欧几里得算法 [2] 分别与  $N$  求最大公约数来找到  $N$  的非平凡因子。如果  $r$  是奇数，那么可以重新选择  $a$ ，因为每次  $r$  为偶数的概率为  $1/2$ ，那么可在有限次重复内找到偶数的  $r$ 。这样因子分解问题就转化为了元素求阶的问题。

考虑如何用量子计算机解决元素求阶问题，首先可以考虑使用传统思路，第一个寄存器  $x$  使用 Hadamard 矩阵构造一个等权叠加态，包含  $0$  到  $N-1$  之间的状态。这样经过一个  $U_f$  变换，第二个寄存器  $y$  就会变为  $|yf(x)\rangle = |f(x)\rangle = |a^x \pmod N\rangle$ 。

现在对第二个寄存器的输出进行测量，假设第二个寄存器塌缩到  $z$ ，那么第一个寄存器就会变成所有满足  $a^l \pmod N = z$  的叠加态，其中  $l$  是满足等式的最小正数，那么寄存器 1 就可以写做

$$|\alpha\rangle = \frac{1}{\sqrt{A+1}} \sum_{j=0}^A |jr + l\rangle \quad (2)$$

其中  $r$  是周期也就是  $a$  的阶，那么对周期函数求周期就可自然想到傅里叶变换，下面将对量子态上的傅里叶变换作介绍。

### 1.3 量子傅里叶变换

首先考虑在经典计算中的傅里叶变换，有一个  $N$  维的矢量， $\{f(0), f(1), \dots, f(N-1)\}$ ，每一个分量  $f(i)$  都是复数，那么傅里叶变换就会把它变成一个新的矢量  $\{\tilde{f}(0), \tilde{f}(1), \dots, \tilde{f}(N-1)\}$ ，其中

$$\tilde{f}(k) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i jk/N} f(j) \quad (3)$$

$\omega = e^{2\pi i/N}$ ，那么公式也可以写为

$$\tilde{f}(k) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \omega^{jk} f(j) \quad (4)$$

类似地，对于量子傅里叶变换，记  $\{|0\rangle \dots |N-1\rangle\}$  是  $N$  维希尔伯特空间中的计算基，量子傅里叶变换  $U_F$  定义为：

$$U_F|j\rangle := \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i jk/N} |k\rangle, j = 0, \dots, N-1. \quad (5)$$

量子傅里叶变换的公式不太直观，下面将在不同角度去理解这件事情。

**从基矢变换角度：**考虑系统中有  $n$  个量子位，那么就有  $2^n$  个基态： $|0\rangle, |1\rangle, \dots, |2^n - 1\rangle$ 。对于量子傅里叶变换就是把原来的基态变成一组新的基态  $|\tilde{y}\rangle$ ，其中：

$$|\tilde{y}\rangle = U_F|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} \omega^{xy} |x\rangle \quad (y = 0, 1, \dots, 2^n - 1) \quad (6)$$

可以使用矩阵来进行更直观表示，例如当  $n=1$  的时候，当  $y=0$  的时候代表基态  $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ ，当  $y=1$  的时候代表基态  $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

$$|\tilde{y}\rangle = \frac{1}{\sqrt{2^n}} \begin{bmatrix} \omega^{0y} \\ \omega^{1y} \\ \vdots \\ \omega^{(2^n-1)y} \end{bmatrix} \quad (7)$$

假设取其中两个基态  $|\tilde{y}_m\rangle$  和  $|\tilde{y}_n\rangle$ ，不难验证当不同基态之间是相互正交的。

$$\begin{aligned} \langle \tilde{y}_m | \tilde{y}_n \rangle &= \frac{1}{N} \begin{bmatrix} \omega^{-0y_m} \omega^{-1y_m} \dots \omega^{-(N-1)y_m} \end{bmatrix} \begin{bmatrix} \omega^{0y_n} \\ \omega^{1y_n} \\ \vdots \\ \omega^{(N-1)y_n} \end{bmatrix} \\ &= \frac{1}{N} \sum_{x=0}^{N-1} \omega^{x(y_n - y_m)} = \begin{cases} 1 & m = n \\ 0 & m \neq n \end{cases} \end{aligned} \quad (8)$$

**从状态变换的角度：**考虑量子系统处于叠加态  $|\phi\rangle = \sum_{j=0}^{2^n-1} f(j)|j\rangle$ ，对它进行量子傅里叶变换可以得到一个新叠加状态

$$|\tilde{\psi}\rangle = U_F|\psi\rangle = \sum_{k=0}^{2^n-1} \tilde{f}(k)|k\rangle \quad (9)$$

其中  $\tilde{f}(k)$  如公式 (3) 所示，那么下一步将通过这个新状态进行周期的提取。

对于量子傅里叶变换的电路表示，是将量子傅里叶变换转化成对每一个量子比特的转化，并将输出态转化为张量积的形式。那么量子电路就从每一个量子比特出发，考虑对每一个量子比特进行转换来构造电路，具体可以参考 Michael A Nielsen and Isaac Chuang 2002 5.1 节。[1]

## 1.4 Shor 算法

有了上文的一些分析，下面来介绍多项式时间求解因式分解问题算法-Shor 算法。

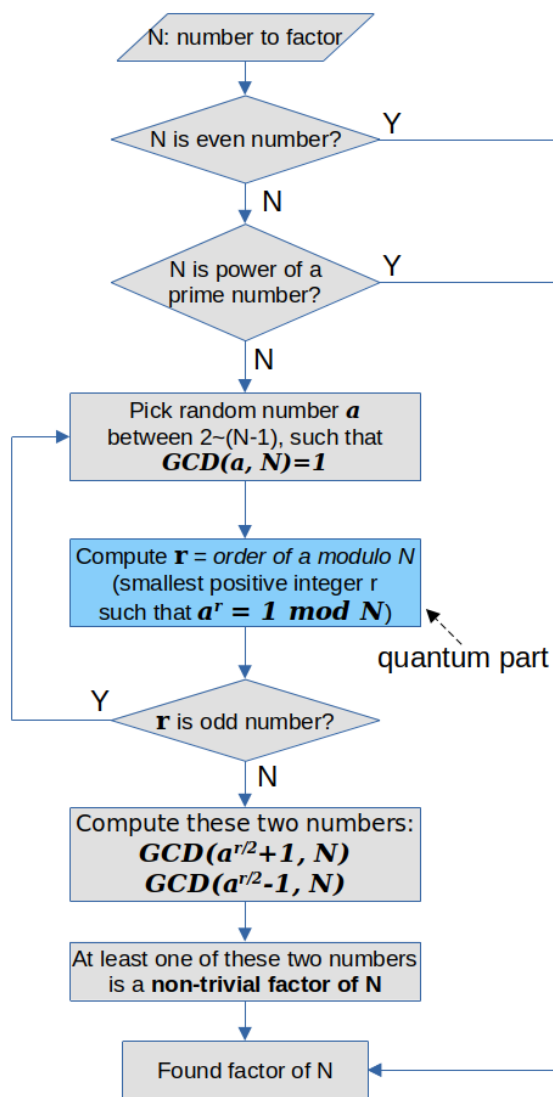


图 2: Shor algorithm

上图所示是 Shor 算法的具体步骤，除蓝色标注部分外，都可以采用经典计算来进行解决，下面将对量子计算的蓝色部分进行详细分析。在 1.2 中已经通过通用量子门得到了一个叠加态  $|\alpha\rangle$ ，下面就是通过量子傅里叶变换提取周期信息。



考虑对状态  $\alpha$  进行傅里叶变换，根据量子傅里叶变换的定义，得到

$$|\tilde{\alpha}\rangle = QFT|\alpha\rangle = \sum_{y=0}^{2^n-1} \tilde{f}(y)|y\rangle \quad (10)$$

其中每一个分量  $\tilde{f}(y)$  是

$$\begin{aligned} \tilde{f}(y) &= \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} \omega^{xy} f(x) \\ &= \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{\frac{2\pi i xy}{N}} f(x) \quad (N = 2^n) \end{aligned} \quad (11)$$

对于  $f(x)$  的取值，只有当  $x \in l, l+r, \dots, l+Ar$ ，或者说当  $x = jr + l$  ( $j = 0, \dots, A+1$ ) 的时候才为  $\frac{1}{\sqrt{A+1}}$ ，其它的分量都为 0。所以说这  $N$  项中，只有  $A+1$  项非 0 项，那么  $\tilde{f}(y)$  可以写为

$$\begin{aligned} \tilde{f}(y) &= \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} e^{\frac{2\pi i xy}{N}} f(x) \\ &= \frac{1}{\sqrt{N}} \left[ \frac{1}{\sqrt{A+1}} \sum_{j=0}^A e^{\frac{2\pi i (jr+l)y}{N}} \right] \\ &= \frac{1}{N(A+1)} e^{2\pi i ly/N} \sum_{j=0}^A e^{2\pi i jry/N} \end{aligned} \quad (12)$$

由于  $e^{2\pi iz}$  对于任意整数  $z$  都成立，所以对于  $e$  的指数，我们只需要关注小数部分，那么  $ry/N$  这个分数，小数部分就可以写成  $\frac{ry \bmod N}{N}$ ，根据连分数定理，在 0 和  $N-1$  之间必然存在某些整数  $y$ ，使得  $ry$  与最近的  $N$  倍数之间的距离小于等于  $r/2$ ，也就是

$$|ry - kN| \leq \frac{r}{2}$$

对两侧同时除以  $rN$ ，可以得到

$$|ry - kN| \leq \frac{r}{2} \Rightarrow \left| \frac{y}{N} - \frac{k}{r} \right| \leq \frac{1}{2N} \quad (13)$$

由于这里  $N = 2^n$ ，因此使用的量子位数量越多，误差也就越小。下面就可以使用连分数算法从  $y/N$  推出和它接近的分数  $k/r$ ，这里面  $r$  就是可能的函数周期。根据 Peter Shor 的论文，我们找到  $r$  的概率至少是  $\delta/\log\log r$ ，其中  $\delta$  是一个常数。因此我们重复算法  $O(\log\log r)$  次，就可以很高概率成功获得  $r$  [3]。Shor 算法能够将因子分解和离散对数困难问题降低到多项式时间，因此基于量子计算机传统的公钥密码算法将被攻破。

## 1.5 算法实例分析及实现

理论分析还是相对比较抽象,下面我们将对一个简单的实例进行分析,能够帮助更好的理解 Shor 算法。

### 分解 $N=15$ :

1. 首先判断  $N$  是一个奇数,若是偶数直接输出因子。
2. 寻找一个与  $15$  互质的数  $a$ , 这里假设我们找到  $13$ 。
3. 运行量子算法寻找周期  $r$ 。

step0:  $15$  可以用  $4$  个比特进行编码,因此我们这里使用  $4$  个量子比特。

step 1: 对第一个寄存器使用 Hadamard 矩阵制备叠加状态,第二个寄存器不做变换

$$[H^{\otimes 4}|0\rangle^{\otimes 4}]|0\rangle^{\otimes 4} = \frac{1}{4} [|0\rangle_4 + |1\rangle_4 + |2\rangle_4 + \dots + |15\rangle_4] |0\rangle_4 \quad (14)$$

step2: 让寄存器 1 和 2 经过  $U_f$  矩阵,寄存器 1 不做变换,寄存器 2 变为  $f(x) \oplus y$ , 其中  $f(x) = a^x \pmod{N}$

$$\begin{aligned} & \frac{1}{4} [|0\rangle_4 |0 \oplus 13^0 \pmod{15}\rangle_4 + |1\rangle_4 |0 \oplus 13^1 \pmod{15}\rangle_4 + \dots] \\ &= \frac{1}{4} [|0\rangle_4 |1\rangle_4 + |1\rangle_4 |13\rangle_4 + |2\rangle_4 |4\rangle_4 + |3\rangle_4 |7\rangle_4 + \dots \\ &= + |12\rangle_4 |1\rangle_4 + |13\rangle_4 |13\rangle_4 + |14\rangle_4 |4\rangle_4 + |15\rangle_4 |7\rangle_4] \end{aligned} \quad (15)$$

step3: 对第二个寄存器进行测量,假设测量的结果为  $7$ , 那么第一个寄存器状态就会塌缩到某些固定状态满足  $a^x = 7 \pmod{15}$

$$\frac{1}{2} [|3\rangle_4 + |7\rangle_4 + |11\rangle_4 + |15\rangle_4] \otimes |7\rangle_4 \quad (16)$$

step4: 对第一个寄存器做量子傅里叶变换,应用公式 (5), 这里我们用的是它的酉矩阵形式

$$\begin{aligned} QFT^\dagger |x\rangle &= \frac{1}{8} \sum_{y=0}^{15} [e^{-\frac{3\pi i y}{8}} |y\rangle + e^{-\frac{7\pi i y}{8}} |y\rangle + e^{-\frac{11\pi i y}{8}} |y\rangle + e^{-\frac{15\pi i y}{8}} |y\rangle] \\ &= \frac{1}{8} [4|0\rangle_4 + 4i|4\rangle_4 - 4|8\rangle_4 - 4i|12\rangle_4] \end{aligned} \quad (17)$$

step5: 测量,对寄存器进行测量,  $1/4$  概率得到各值。

- a. 得到  $0$ , 无意义

b. 得到 4,  $y/N=1/4$ , 则  $k=1$ ,  $r=4$

$$x \equiv a \pmod{N} = 13^2 \equiv 4 \pmod{15}$$

$$x + 1 = 5; \gcd(x + 1, N) = 5$$

$$x - 1 = 3; \gcd(x - 1, N) = 3$$

c. 得到 8,  $y/N = 1/2$ , 则  $k = 1, r = 2$  或者  $k = 2, r = 4$  但是当  $r = 2$  只能得到部分根

$$13^1 \equiv 13 \pmod{15}$$

$$x + 1 = 14; \gcd(x + 1, N) = 1$$

$$x - 1 = 12; \gcd(x - 1, N) = 3$$

d. 得到 12,  $y/N = 3/4$ , 则  $k = 3, r = 4$

我们尝试使用 Qiskit 平台对 Shor 算法进行实现, 最终成功得到了对  $r$  的估计和 15 的因子。具体代码和注释将在附录中给出

	Phase	Fraction	Guess for $r$
0	0.50	1/2	2
1	0.25	1/4	4
2	0.00	0/1	1
3	0.75	3/4	4

The possible result of 15:  
[3, 5]

图 3: Result of the code

## 1.6 Shor 算法的改进

对于 Shor 算法有多种改进, 比如在连分数算法提升速率, 或者采用分组优化等, 下面我们将介绍一种对于周期估计的优化。具体来讲 Shor 算法中  $r$  的估计必须是偶数, 这样有 0.5 的概率算法重复执行, 下面将条件放宽, 在 3 的倍数情况下将  $a^r - 1 = 0 \pmod{N}$ , 分解为  $a^{\frac{2r}{3}} + a^{\frac{r}{3}} + 1$  和  $a^{\frac{r}{3}} - 1$ , 当然这两个数必须分别满足模  $N$  情况下不为 0, 然后就可以分别和  $N$  计算最大公因子来求  $N$  的非平凡因子, 这样的情况下使算法重复执行的概率下降

到 0.33。当然更一般地，可以将  $a^r - 1$  分解为  $a - 1$  和  $a^{r-1} + a^{r-2} + \dots + 1$ ，但在这种情况下得到素因子的概率很低且计算量很大 [4]。

## 2 后量子密码学

上面我们通过 shor 算法展现了量子计算在解决密码问题中的强大威力，另外对于对称密码，也可以采用 Grover 算法等相应的量子算法极大的减少时间复杂度 [5]。为了对抗量子计算，采用格困难问题的后量子密码应运而生。

### 2.1 LWE 问题

LWE(Learning With Error) 问题是格中的一类困难问题，简单来说就是给定一个矩阵  $A$  和一个向量  $\hat{b}$ ，并且加入一个随机噪声向量  $e$ ，来还原未知数向量  $x$ 。

$$\hat{b} = Ax + e \quad (18)$$

具体来说 LWE 分为搜索 LWE 问题 (SLWE) 和决策 LWE 问题 (DLWE)。

**搜索 LWE 问题:**

$LWE(n, m, q, x_B)$  Search Version

$$A \xleftarrow{R} \mathbb{Z}_q^{m \times n}, s \xleftarrow{R} \mathbb{Z}_q^n, e \xleftarrow{R} x_B^m$$

Given  $(A, As + e)$ , find  $s' \in \mathbb{Z}_q^n$  s.t.  $\|As' - (As + e)\|_\infty \leq B$

简单来说，也就是在  $\mathbb{Z}_q$  素数有限域上给定矩阵  $A$  以及随机向量  $s$  和随机误差  $e$ ，找到  $As'$ ，使得得到的向量和  $As + e$  之间误差不能超过上界  $B$ 。

- $n$  为 LWE 问题的安全参数， $n$  越大代表变量数目越多，LWE 问题越困难
- $m$  一般为  $n$  的多项式倍数， $m$  越多代表方程组越多，LWE 问题越简单
- $q$ ，是有限域的大小，为了密码学应用和计算方便，使用  $\mathbb{Z}_q$  素数有限域
- $B$  误差上界，决定问题中解与实际取值  $\hat{b}$  之间的距离

## 决策 LWE 问题:

$LWE(n, m, q, x_B) : \text{Decisional Version}$

$$A \xleftarrow{R} \mathbb{Z}_q^{m \times n}, s \xleftarrow{R} \mathbb{Z}_q^n, e \xleftarrow{R} x_B^m, v \xleftarrow{R} \mathbb{Z}_q^m$$

Distinguish  $(A, As + e)$  from  $(A, v)$

在密码学中，一般需要证明一个困难问题安全性时，一般使用决策版本 LWE 问题，也就是说给出  $A, \hat{b}$ ，辨别是一个 LWE 问题实例还是一个随机向量  $v$ 。

## 2.2 Regev 公钥加密算法

基于 LWE 困难问题，Regev 提出了 Regev 公钥加密算法。对于参数的选取，一般是先选择一个安全参数  $n$ ，然后将  $m, q, B$  都设置成一个函数  $f(n)$  的输出。下面来具体进行介绍：

**-Private Key:** 在  $\mathbb{Z}_q$  随机选择一个  $s$ ,  $s \xleftarrow{R} \mathbb{Z}_q^n$

**-Public Key:**  $A \xleftarrow{R} \mathbb{Z}_q^{m \times n}, e \xleftarrow{R} x_B^m$ , 计算  $b = As + e$ 。其中向量  $e$  中每一个元素都小于上界  $B$ ,  $B$  满足  $q/4 > mB$ ,  $pk = (A, b)$

**-Encryption:** 对每一个二进制位进行单独加密，首先选择一个随机向量  $r \xleftarrow{R} \mathbb{Z}_2^m \in \{0, 1\}^m$ ，然后计算密文的第一部分  $c_0 \leftarrow r^T A$ ，然后计算密文的第二部分  $c_1 \leftarrow r^T b + \lfloor q/2 \rfloor x$ ，最后输出  $(c_0, c_1)$

**-Decryption:** 计算  $\hat{x} = c_1 - c_0 s$ ，若结果的绝对值  $|\hat{x}| < q/4$ ，那么输出  $x = 0$ ，否则输出  $x = 1$ 。

### 2.2.1 Regev 加密的正确性

对于 Regev 加密的正确性，我们可以将解密部分进行展开

$$\begin{aligned} \hat{x} &= c_1 - c_0 \cdot s \\ &= r^T b + q/2 \cdot x - r^T A s \\ &= r^T (A s + e) - r^T A s + q/2 \cdot x \\ &= r^T e + q/2 \cdot x \end{aligned}$$

可以看到最终解密的结果是原文比特乘  $q/2$ ，然后加上了  $r^T e$  的误差噪声，其中  $r$  能够将误差的范围限制到  $B$  以内。这里注意我们在运算过程中采用

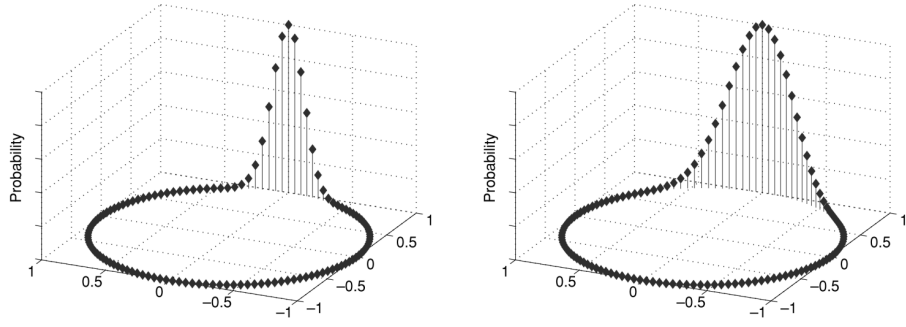


图 4: The elements of  $\mathbb{Z}_p$  are arranged on a circle.

的是中心模，也就是  $a \bmod p = a - [a/p]p$ ，可以看作将  $(p/2, p]$  映射到  $(-p/2, 0]$  上。

通过 Regev 论文给出图 4 [6]，明文比特被映射到  $(-q/2, q/2]$  这个有限环域上，0 代表环的一侧，1 代表环的另一侧 ( $q/2$  位置)，随机噪声就相当于在实际位置增加了一些扰动。当然如果扰动过大，实际值到达另一侧则实际比特就无法辨别，因此规定  $q/4 > mB$ 。

### 2.2.2 Regev 加密的安全性

下面将使用混合论证来证明 Regev 加密体系是语义安全的，首先假设一个第三方能够看到 Regev 加密密文的所有信息，那么他可以得到

$$H_0: pk = (A, \hat{b} = As + e), c_0 = r^T A, c_1 = r^T b + q/2 \cdot x \quad (19)$$

下面根据 DLWE 的假设，第三方无法分别 LWE 实例和随机向量，我们可以将  $As + e$  替换为  $v$ ，那么我们可以构造

$$H_1: pk = (A, v), c_0 = r^T A, c_1 = r^T b + q/2 \cdot x \quad (20)$$

接着，我们可以将所有非随机参数替换为随机向量，得到

$$H_2: pk = (A, v), c_0 \xleftarrow{R} \mathbb{Z}_q^n, c_1 \xleftarrow{R} \mathbb{Z}_q \quad (21)$$

根据剩余哈希定理 (Leftover Hash Lemma)，我们可以认为  $H_1$  和  $H_2$  是无法被第三方辨别的。另外，根据 DLWE 假设，第三方也无法分辨  $H_0$  和  $H_1$ ，因此第三方也无法分辨  $H_0$  和  $H_2$ ，而  $H_2$  中没有任何对于原文  $x$  加密的信息，因此可以认为 Regev 加密是语义安全的。

### 2.2.3 Regev 加密的实现

下面尝试使用 python 对 Regev 公钥加密算法进行实现, 其中实现过程中需要注意下面的几点, 具体细节在附录中给出:

1. 由于 Regev 加密是对单个比特进行操作, 因此首先将字符串转化为二进制。
2. 在生成噪声  $e$  的时候注意  $e < \frac{q}{4m}$ 。
3. 在进行加密前时对明文二进制串进行补全, 使维度匹配。

```
In [19]: #首先生成公私钥对
sk, pK=Keygen()
#进行明文的填充
print("加密信息: ", str)
str = extend("Hello world!")
#对填充后的明文用公钥进行加密
c = encrypt(pK, str)
#print("密文: ", c)
#对密文用私钥进行解密
b_str = decrypt(sk, m)
#转化为字符串打印
print("解密结果:", ToStr(b_str))

加密信息: Hello world!
解密结果: Hello world!
```

图 5: Regev 公钥加密实现结果

## 2.3 LWE 与全同态加密

另外 LWE 问题在构造全同态加密方面也有强大的作用, Brakerski 和 Vaikuntanathan 在 2011 年提出了基于 LWE 的有限级数的同态加密系统, 并可以通过 Bootstrapping 的方式类变成全同态加密 [7]。另外 Gentry, Sahai 和 Waters 在 2013 年也基于 LWE 假设, 提出了第三代的全同态加密系统 [8], 该方案的同态加法和同态乘法通过做简单的矩阵加法和乘法来实现, 从而使得 GSW-FHE 方案简单快速, 并且同态运算不需要使用计算公钥, 只需要借助用户的公钥即可实现。

## 参考文献

- [1] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.
- [2] Donald Knuth. The art of computer programming, 2 (seminumerical algorithms). (*No Title*), 1981.
- [3] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.
- [4] 王平平, 陆正福, 杨春尧, and 李军. Shor 量子算法的分析及优化. 通信技术, 50(4):775–778, 2017.
- [5] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.
- [6] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
- [7] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.
- [8] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.

## 附录 A Regev 公钥加密算法

```
1 #首先将字符串转化为二进制
2 def BinChange(s):
3     b_list = []
4     for c in s:
5         c = bin(ord(c)).replace('0b', '')
6         if len(c) <= 8:
7             b_list.append(c.zfill(8)) #将二进制串补为8个bit
8     return b_list
```



```

9
10 #对二进制串进行扩展,  $r^Tb, b=As+e, b(m,1)$ , 需要m维
11 def extend(str):
12     length = len(str)
13     #print(length)
14     #print(M)
15     if (length*8<M):
16         for i in range(0,(int(M/8)-length)):
17             str += " "
18     #print(str)
19     return str
20
21 #key生成过程注意噪声e的范围
22 def Keygen():
23     s = np.random.randint(0,q,[N,1])
24     A = np.random.randint(0,q,[M,N])
25     e = np.random.randint(0,math.ceil(q/(4*M))-1,[M,1])
26     #设置参数
27     b = (np.matmul(A,s)+e)%q
28     return s,(A,b)
29
30 #加密过程
31 def encrypt(pK,str):
32     b_list = np.array(list(''.join(BinChange(str))), dtype=np.int32).reshape(M, 1)
33     r = np.random.randint(0,1,[M,1])
34     c0 = np.dot(r.T,pK[0])
35     c1 = np.dot(r.T,pK[1])+math.ceil(q/2)*b_list
36     return (c0,c1)
37
38 #解密过程, 需要判断x在环上的位置
39 def decrypt(sk,m):
40     b_str = ""
41     c0 = m[0]
42     c1 = m[1]
43     x = c1-np.dot(c0,sk)

```

```

44     x.reshape(1,-1)
45     for i in x:
46         if (i<(q/4)):
47             b_str += "0"
48         else:
49             b_str += "1"
50     return b_str
51
52 #将二进制串转化为字符串
53 def ToStr(b_str):
54     result = ""
55     length = int(len(b_str)/8)
56     for i in range(0,length):
57         temp = "0b"+b_str[i*8:(i+1)*8]
58         result +=(chr(int(temp,2)))
59     return result

```

## 附录 B Qiskit Shor algorithm

```

1 from qiskit import QuantumCircuit, Aer, transpile
2 from qiskit.visualization import plot_histogram
3 from math import gcd
4 from numpy.random import randint
5 import pandas as pd
6
7 #a mod 15
8 def c_amod15(a, power):
9     """Controlled multiplication by a mod 15"""
10    if a not in [2,4,7,8,11,13]:
11        raise ValueError("'a' must be 2,4,7,8,11 or 13")
12    U = QuantumCircuit(4)
13    for __iteration in range(power):
14        if a in [2,13]:
15            U.swap(2,3)
16            U.swap(1,2)

```

```

17         U.swap(0,1)
18         if a in [7,8]:
19             U.swap(0,1)
20             U.swap(1,2)
21             U.swap(2,3)
22         if a in [4, 11]:
23             U.swap(1,3)
24             U.swap(0,2)
25         if a in [7,11,13]:
26             for q in range(4):
27                 U.x(q)
28     U = U.to_gate()
29     U.name = f" $\{a\}^{\{power\}} \bmod 15$ "
30     c_U = U.control()
31     return c_U
32
33 #量子傅里叶逆运算
34 def qft_dagger(n):
35     """n-qubit QFTdagger the first n qubits in circ"""
36     qc = QuantumCircuit(n)
37     # Don't forget the Swaps!
38     for qubit in range(n//2):
39         qc.swap(qubit, n-qubit-1)
40     for j in range(n):
41         for m in range(j):
42             qc.cp(-np.pi/float(2**(j-m)), m, j)
43         qc.h(j)
44     qc.name = "QFT†"
45     return qc
46
47 # Create QuantumCircuit with N_COUNT counting qubits
48 # plus 4 qubits for U to act on
49 qc = QuantumCircuit(N_COUNT + 4, N_COUNT)
50
51 # Initialize counting qubits

```

```

52 # in state |+>
53 for q in range(N_COUNT):
54     qc.h(q)
55
56 # And auxiliary register in state |1>
57 qc.x(N_COUNT)
58
59 # Do controlled-U operations
60 for q in range(N_COUNT):
61     qc.append(c_amod15(a, 2**q),
62               [q] + [i+N_COUNT for i in range(4)])
63
64 # Do inverse-QFT
65 qc.append(qft_dagger(N_COUNT), range(N_COUNT))
66
67 # Measure circuit
68 qc.measure(range(N_COUNT), range(N_COUNT))
69 qc.draw(fold=-1) # -1 means 'do not fold'
70
71 rows, measured_phases = [], []
72 for output in counts:
73     decimal = int(output, 2) # Convert (base 2) string to decimal
74     phase = decimal/(2**N_COUNT) # Find corresponding eigenvalue
75     measured_phases.append(phase)
76     # Add these values to the rows in our table:
77     rows.append([f"{output}(bin) = {decimal:>3}(dec)",
78                 f"{decimal}/{2**N_COUNT} = {phase:.2f}"])
79 # Print the rows in a table
80 headers=["Register Output", "Phase"]
81 df = pd.DataFrame(rows, columns=headers)
82 print(df)
83 rows = []
84 for phase in measured_phases:
85     frac = Fraction(phase).limit_denominator(15)
86     rows.append([phase,

```

```

87         f"{frac.numerator}/{frac.denominator}",
88         frac.denominator])
89 # Print as a table
90 headers=["Phase", "Fraction", "Guess for r"]
91 df = pd.DataFrame(rows, columns=headers)
92 print(df)
93
94
95
96 def qpe_amod15(a):
97     """Performs quantum phase estimation on the operation a*r mod 15.
98     Args:
99         a (int): This is 'a' in a*r mod 15
100     Returns:
101         float : Estimate of the phase
102     """
103     N_COUNT = 8
104     qc = QuantumCircuit(4+N_COUNT, N_COUNT)
105     for q in range(N_COUNT):
106         qc.h(q) # Initialize counting qubits in state |+>
107         qc.x(3+N_COUNT) # And auxiliary register in state |1>
108         for q in range(N_COUNT): # Do controlled-U operations
109             qc.append(c_amod15(a, 2**q),
110                       [q] + [i+N_COUNT for i in range(4)])
111         qc.append(qft_dagger(N_COUNT), range(N_COUNT)) # Do inverse-QFT
112         qc.measure(range(N_COUNT), range(N_COUNT))
113     # Simulate Results
114     aer_sim = Aer.get_backend('aer_simulator')
115     # 'memory=True' tells the backend to save each measurement in a list
116     job = aer_sim.run(transpile(qc, aer_sim), shots=1, memory=True)
117     readings = job.result().get_memory()
118     print("Register Reading: " + readings[0])
119     phase = int(readings[0],2)/(2**N_COUNT)
120     print(f"Corresponding Phase: {phase}")
121     return phase

```

```
122
123 #Get the solution
124 phase = qpe_amod15(a) # Phase = s/r
125 Fraction(phase).limit_denominator(15)
126 frac = Fraction(phase).limit_denominator(15)
127 s, r = frac.numerator, frac.denominator
128 print(r)
129 guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
130 print(guesses)
131 a = 7
132 FACTOR_FOUND = False
133 ATTEMPT = 0
134 while not FACTOR_FOUND:
135     ATTEMPT += 1
136     print(f"\nATTEMPT {ATTEMPT}:")
137     phase = qpe_amod15(a) # Phase = s/r
138     frac = Fraction(phase).limit_denominator(N)
139     r = frac.denominator
140     print(f"Result: r = {r}")
141     if phase != 0:
142         # Guesses for factors are  $\gcd(x^{\{r/2\}} \pm 1, 15)$ 
143         guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
144         print(f"Guessed Factors: {guesses[0]} and {guesses[1]}")
145         for guess in guesses:
146             if guess not in [1, N] and (N % guess) == 0:
147                 # Guess is a factor!
148                 print("*** Non-trivial factor found: {guess} ***")
149                 FACTOR_FOUND = True
```