Analysis the decryption failure probability based on LWE schemes

基于格上困难问题设计的公钥密码方案被认为是能够抵抗量子攻击的。多数格密码方案都是基于 2011 年 Linder 和 Peikert 提出的 LP 方案 [7] 的变体，如 Frodo, NewHope, LAC, Kyber 等。但是基于 LP 方案构造的密码方案可能会出现解密错误的情况，敌手可以利用解密错误的得到私钥的一些信息 [6]。因此在设计基于 LP 方案设计公钥加密方案时，分析并控制解密错误的概率是十分重要的。本文将首先介绍 LP 方案的基本方法，再依次介绍基于 LWE(Frodo), RLWE(LAC), MLWE(Kyber),MLWR(Saber) 三种不同方案解密错误概率的计算。

1 LP 方案与概率分析的两个公式

1.1 LWE 问题与 LWR 问题

LP 方案的困难性假设都来自于 LWE(Learing with errors) [9] 问题及其变体, 其中 LWR(Learing with rounding) [1] 问题也是 LWE 问题的变种。将原本在 $\mathbb{Z}_q^n$ 中的运算扩展到 R_q、R_q^n 中构成了 R(Ring)/M(Mod) LWE(/R) 问题。

LWE 问题分为搜索 LWE 与判定 LWE 问题，LWR 问题也可分为这两种问题。

定义 1 (搜索 LWE 问题) 取 n, m, q 为正整数，χ_s, χ_e 为 $\mathbb{Z}$ 上的分布，给定实列 $(A, b = AS + e)$, 求密钥向量 s, 其中 $A \xleftarrow{\$} \mathbb{Z}_q^{m\times n}$, 密钥向量 $s \xleftarrow{\$} \chi_s^n$，错误向量 $e \xleftarrow{\$} \chi_e^m$。

定义 2 (判定 LWE 问题) 取 n, m, q 为正整数，χ_s, χ_e 为 $\mathbb{Z}$ 上的分布, 区分以下两种分布:

分布 1(A, b), 其中 $b = As + e, A \xleftarrow{\$} \mathbb{Z}_q^{m\times n}, s \xleftarrow{\$} \chi_s^n, e \xleftarrow{\$} \chi_e^m$

分布 2(A, u), 其中 $A \xleftarrow{\$} \mathbb{Z}_q^{m\times n}, u \xleftarrow{\$} \chi_s^m$

为描述 LWR 问题，定义运算 $\lfloor x \rceil_{q\to p} = \lfloor \frac{p}{q} \cdot x \rceil \bmod p$

定义 3 (搜索 LWR 问题) 取 n, p, q 为正整数且 $q \geqslant p \geqslant 2$。给定 $(a, b = \lfloor \langle a, s\rangle \rceil_{q\to p} \in \mathbb{Z}_q^n \times \mathbb{Z}_p)$, 求密钥 s, 其中 $a \xleftarrow{\$} \mathbb{Z}_q^n, s \xleftarrow{\$} \mathbb{Z}_q^n$。

定义 4 (判定 LWR 问题) 取 n, p, q 为正整数且 $q \geqslant p \geqslant 2$，区分以下两种分布

分布 1(a, b), 其中 $b = \lfloor \langle a, s\rangle \rceil_{q\to p}, a \xleftarrow{\$} \mathbb{Z}_q^n, s \xleftarrow{\$} \mathbb{Z}_q^n$

分布 2(a, u), 其中 $a \xleftarrow{\$} \mathbb{Z}_q^n, u \xleftarrow{\$} \mathbb{Z}_p^n$

1.2 LP 方案

LP 方案会出现解密错误的本质在于对消息的编码与解码。对于比特长度为 ℓ 的消息 $m \in \{1,0\}^{\ell}$，定义编码函数 $\widetilde{m}[i] = encode(m[i]) = m[i] \cdot \lfloor \frac{q}{2} \rfloor$，解码函数

$$m[i] = decode(\widetilde{m}[i]) = \begin{cases} 0, & \text{if } \widetilde{m}[i] \in [-\lfloor \frac{q}{4} \rfloor, \lfloor \frac{q}{4} \rfloor) \\ 1, & \text{else} \end{cases}$$

即解码函数能够正确恢复出带有错误值 $e(||e|| < \lfloor \frac{q}{4} \rfloor)$ 的消息，其中 q 为具体方案选择的模数。

以下介绍 LP 方案中密钥生成、加密、解密的具体步骤。

$\mathrm{Gen}(\overline{\mathbf{A}} \in \mathbb{Z}_q^{n_1 \times n_2}, \ell)$:

选择 $\mathbf{R}_1 \leftarrow D_{\mathbb{Z},s_k}^{n_1 \times \ell}$, $\mathbf{R}_2 \leftarrow D_{\mathbb{Z},s_k}^{n_2 \times \ell}$，计算 $\mathbf{P} = \mathbf{R}_1 - \overline{\mathbf{A}} \cdot \mathbf{R}_2 \in \mathbb{Z}_q^{n_1 \times \ell}$。公钥为 $\overline{\mathbf{A}}, \mathbf{P}$, 私钥为 $\mathbf{R}_2$。公钥与私钥间的关系按照矩阵表示如下：

$$\begin{bmatrix} \overline{\mathbf{A}} & \mathbf{P} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_2 \\ \mathbf{I} \end{bmatrix} = \mathbf{R}_1 \bmod q$$

其中 $\mathbf{R} \leftarrow D_{\mathbb{Z},s_k}^{n \times m}$ 表示 $\mathbf{R}$ 是从在整数上的分布 $D_{\mathbb{Z}}$ 中以参数 s_k 采样而来，并构成 $n \times m$ 维的矩阵。

$\mathrm{Enc}(\overline{\mathbf{A}}, \mathbf{P}, m \in \{0,1\}^{\ell})$:

选择从分布 $D_{\mathbb{Z},s_e}$ 中独立选取 $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3) \in \mathbb{Z}^{n_1} \times \mathbb{Z}^{n_2} \times \mathbb{Z}^{\ell}$，将消息 m 编码计算 $\widetilde{m} = encode(m) \in \mathbb{Z}_q^{\ell}$，计算密文

$$\mathbf{c}^t = \begin{bmatrix} \mathbf{c}_1^t & \mathbf{c}_2^t \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^t & \mathbf{e}_2^t & \mathbf{e}_3^t + \overline{\mathbf{m}}^t \end{bmatrix} \cdot \begin{bmatrix} \overline{\mathbf{A}} & \mathbf{P} \\ \mathbf{I} & \\ & \mathbf{I} \end{bmatrix} \in \mathbb{Z}_q^{1 \times (n_2 + \ell)}$$

即 $c_1^t = e_1^t \overline{\mathbf{A}} + e_2^t$，$c_2^t = e_1^t \mathbf{P} + e_3^t + \widetilde{m}^t = e_1^t \mathbf{R}_1 - e_1^t \overline{\mathbf{A}} \cdot \mathbf{R}_2 + e_3^t + \widetilde{m}^t$

$\mathrm{Dec}(\mathbf{c}^t = [\mathbf{c}_1^t, \mathbf{c}_2^t], \mathbf{R}_2)$:

使用密钥生成与加密函数中的关系，计算

$$\widetilde{m}^t = \begin{bmatrix} \mathbf{c}_1^t & \mathbf{c}_2^t \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_2 \\ \mathbf{I} \end{bmatrix} = \left(\mathbf{e}^t + \begin{bmatrix} \mathbf{0} & \mathbf{0} & \overline{\mathbf{m}}^t \end{bmatrix} \right) \cdot \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ \mathbf{I} \end{bmatrix} = \mathbf{e}^t \cdot \mathbf{R} + \overline{\mathbf{m}}^t$$

其中 $\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ \mathbf{I} \end{bmatrix}$，即 $\widetilde{m} = e_1^t R_1^t + e_2^t R_2^t + e_3^t + \overline{\mathbf{m}}^t$。若 $||e_t \cdot \mathbf{R}||_{\infty} < \frac{q}{4}$，则 $decode$ 函数可正确恢复消息 m。

因此，分析 LP 方案解密错误概率的核心在于根据不同体制选取的错误分布计算冗余项的概率分布。

1.3 概率分析的两个公式

根据 1.1 节的分析，在计算冗余项的概率分布时需要用到乘法公式与卷积公式。

乘法公式：对于事件 A, B，两事件同时发生 AB 的概率 $P(AB) = P(A)P(B|A) = P(B)P(A|B)$，当 A, B 相互独立时 $P(AB) = P(A)P(B)$。

卷积公式：对于两随机变量 X, Y，且 X, Y 相互独立，若变量 $Z = X + Y$，则 $P\{Z = z\} = P\{X + Y = z\} = \sum_{i=0}^{r} x_i y_{r-i}$

2 Frodo: 基于 LWE 方案的错误概率分析

2.1 Frodo 加密方案的基本函数

Frodo [2] 是基于 LWE 构造的加密方案。以下是对 Frodd 第二轮提交方案的抽象描述，并不涉及具体实现。

Frodo.KG():

$$1.A \xleftarrow{\$} U(\mathbb{Z}_q^{n\times n})$$
$$2.S^T \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{\overline{n}\times n}$$
$$3.E \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{n\times \overline{n}}$$
$$4.B = AS + E \in \mathbb{Z}_q^{n\times \overline{n}}$$
$$5.\text{return } pk = (A, B), sk = S$$

Frodo.Enc(pk, u):

$$1.S' \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{\overline{n}\times n}$$
$$2.E' \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{\overline{n}\times n}$$
$$3.E'' \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{\overline{n}\times \overline{n}}$$
$$4.B' = S'A + E' \in \mathbb{Z}_q^{\overline{n}\times n}$$
$$5.V = S'B + E'' + \text{Encode}(u) \in \mathbb{Z}_q^{\overline{n}\times \overline{n}}$$
$$6.\text{return } c = (B', V)$$

Frodo.Dec(sk, c):

$$1.m = V - B'S$$
$$2.\text{return Decode}(m)$$

其中 $A \xleftarrow{\$} U(\mathbb{Z}_q^{n\times n})$ 表示 A 是从 $\mathbb{Z}_q$ 中均匀选取的 $n \times n$ 维矩阵。$E \xleftarrow{\$} \mathbb{Z}_{q,\chi}^{n\times \overline{n}}$ 表示 E 是按照分布 χ 从 $\mathbb{Z}_q$ 中选取的 $n \times \overline{n}$ 维矩阵。

编码函数 Encode(m) 是对比特长度为 $l = B \cdot \overline{n}^2$ 的消息进行编码，并形成一个 $\overline{n} \times \overline{n}$ 的矩阵。具体为: 依次从低位到高位读取 B 个比特的消息，并将其与整数 $\frac{q}{2^B}$ 相乘，其中 $q = 2^D$。

解码函数 Decode(m) 是编码函数的逆过程，将 $\overline{n} \times \overline{n}$ 的矩阵解码为长为 $l = B \cdot \overline{n}^2$ 的消息。具体为: 对矩阵中的任何一项 C_{ij} 都有 $C'_{ij} = \lfloor \frac{C_{ij}+2^{D-B-1}}{2^{D-B}} \rfloor$。

2.2 Frodo 解密错误概率分析

根据上述描述的解密过程，$m = V - B'S = S'E - E'S + E'' + \text{Encode}(u)$。由 2.1 中定义的编码与解码过程，当 $||W_{ij}|| = ||(S'E - E'S + E'')_{ij}|| < 2^{D-B-1}$ 时，能够正确解密。

$S'E$ 与 $E'S$ 中单项的分布都是 $\chi \times \chi$，由于进行了矩阵乘法操作，$S'E$ 与 $E'S$ 的完整分布都为单项分布自身进行 n 次卷积操作，之后便易于得到 W 的概率分布。对 W 分布中大于 2^{D-B-1} 的概率进行求和便能够得到解密出错的概率。

3 LAC: 基于 RLWE 方案的错误概率分析

3.1 LAC 加密方案的基本函数

LAC [8] 是基于 RLWE 构造的加密方案，在该加密方案中，由于选择的模数 q 较小，导致错误概率增大到不可接受，因此在该方案中使用了纠错码 (ECC, error correction code) 的技术来降低出错概率。

以下算法是对 LAC 第一轮提交算法的抽象描述，并不涉及具体实现。

LAC.KG():

$$
\begin{aligned}
&1.a \xleftarrow{\$} U(R_q)\\
&2.\text{s} \xleftarrow{\$} \Psi_\sigma^n\\
&3.\text{e} \xleftarrow{\$} \Psi_\sigma^n\\
&4.b \leftarrow as + e \in R_q\\
&5.\text{return} \quad pk = (a, b), sk = s
\end{aligned}
$$

LAC.Enc(a, b, m):

$$
\begin{aligned}
&1.\widehat{m} \leftarrow \text{ECCEnc}(m) \in \{0,1\}^{l_v}\\
&2.(r, e_1, e_2) \leftarrow (\Psi_\sigma^n, \Psi_\sigma^n, \Psi_\sigma^{l_v})\\
&3.c_1 \leftarrow ar + e_1 \in R_q\\
&4.c_2 \leftarrow (br)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m} \in \mathbb{Z}_q^{l_v}\\
&5.\text{return} \quad c = (c_1, c_2) \in R_q \times \mathbb{Z}_q^{l_v}
\end{aligned}
$$

LAC.Dec(sk, c):

$$1.u \leftarrow c_1 s \in R_q$$
$$2.\widehat{m'} \leftarrow c_2 - (u)_{l_v} \in \mathbb{Z}_q^{l_v}$$
$$3.\widehat{m} = decode(\widehat{m'})$$
$$4.\text{return} \quad m = \text{ECCDec}(\widehat{m})$$

以上算法中 $R_q = \mathbb{Z}_q[x]/(x^n+1)$, $U(R_q)$ 表示 R_q 上的均匀分布。概率分布 Ψ_σ 中 $\sigma \in \{1, \frac{1}{2}\}$，分别表示分布:

$$\Psi_1 : P(x=0) = \frac{1}{2}, P(x=\pm 1) = \frac{1}{4}$$
$$\Psi_{\frac{1}{2}} : P(x=0) = \frac{3}{4}, P(x=\pm 1) = \frac{1}{8}$$

Ψ_σ^n 表示独立采样 n 次。

3.2 LAC 解密错误概率分析

根据上述算法描述解密过程中 $\widehat{m'} = (er - e_1 s)_{l_v} + e_2 + \lfloor \frac{q}{2} \rceil \cdot \widehat{m}$, 因此 LAC 解密错误的概率来自 $w = (er - e_1 s)_{l_v} + e_2$。可以认为 e, r, e_1, s 两两间相互独立，且每一个系数都来自于分布 Ψ_σ，则 $P(e[i]r[j]) = P(e[i])P(r[j]) = P(\Psi_\sigma \times \Psi_\sigma)$, $P(e_1[i]s[j]) = P(e_1[i])P(s[j]) = P(\Psi_\sigma \times \Psi_\sigma)$。

对于在环 R_q 上的两多项式 $f(x) = \sum_{i=0}^{n-1} f_i x^i, g(x) = \sum_{i=0}^{n-1} g_i x^i$, 则两多项式的乘法为 $h(x) = f(x)g(x) = \sum_{k=0}^{n-1} \left(\left(\sum_{i=0}^{k} f_i g_{k-i} - \sum_{i=k+1}^{n-1} f_i g_{k+n-i} \right) mod q \right) x^k$。即 x^k 的系数为 $\left(\sum_{i=0}^{k} f_i g_{k-i} - \sum_{i=k+1}^{n-1} f_i g_{k+n-i} \right)$，共需 n 次减法运算。由于选择的概率分布 Ψ_σ 都是中心分布 (平均值为 0，关于原点对称)，在计算概率时加法与减法运算并无区别。因此在计算 er 的第 i 项系数的概率分布 $P(er[i])$ 时需要对 $P(\Psi_\sigma \times \Psi_\sigma)$ 自身进行 n 次卷积操作。

由上述分析可以计算出 w 第 i 项系数的概率分布，那么一个比特出现错误的概率 $\delta = 1 - P(-\lfloor \frac{q}{4} \rceil < w_i < \lfloor \frac{q}{4} \rceil)$。

而 LAC 体制中使用了纠错码，使得该体制能够接受 l_t 个比特出现错误并纠正，即当错误比特数大于等于 $l_t + 1$ 时，会出现解密错误。共有 l_v 个比特，每个比特出现错误的概率都为 δ，恰有 k 个比特出现错误，该模型为伯努利概型。因此，LAC 解密错误的概率为:

$$\Delta \approx \sum_{j=l_t+1}^{l_v} \left(\binom{l_v}{j} \delta^j (1-\delta)^{l_v - j} \right)$$

上述模型仅适用于各比特出错概率相互独立的情形，文献中 [5] 指出由于纠错码的使用，导致一个比特出现错误会影响到其他比特出现错误的概率增加，即上述模型会低

估整体出现错误的概率。具体表现为一个错误向量的二范数过大时，将会影响增比特间的相关性，导致解密错误概率增加。

为规避上述问题，在 LAC 提交的第二轮方案中，设计者定义了固定汉明重量的中心二项分布 $\Psi_{\sigma}^{n,h}$，其中 $0 < h < \frac{n}{2}$，且 h 为偶数。该分布输出 1 和-1 的个数为 $\frac{h}{2}$，输出 0 的个数为 $n-h$。该分布能够控制输出 $1, -1$ 的个数，即输出的汉明重量，从而达到控制错误向量范数的目的，保证比特间的独立性，降低解密错误概率。

4 Kyber: 基于 MLWE 方案的错误概率分析

4.1 Kyber 加密方案的基本函数

Kyber [3] 是基于 MLWE 构造的加密方案，在该方案中用到了压缩函数，能够降低传输带宽，但也会损失少许信息。以下为 Kyber 第三轮提交算法的抽象描述，并不涉及具体实现。

Kyber.KG():

$$1.A \xleftarrow{\$} U(R_q^{k\times k})$$
$$2.s \xleftarrow{\$} R_{q,\eta_1}^k$$
$$3.e \xleftarrow{\$} R_{q,\eta_1}^k$$
$$4.t = As + e$$
$$5.\text{return } (pk = (A,t), sk = s)$$

Kyber.Enc($pk, m \in R_q$):

$$1.r \xleftarrow{\$} R_{q,\eta_1}^k$$
$$2.e_1 \xleftarrow{\$} R_{q,\eta_2}^k$$
$$3.e_2 \xleftarrow{\$} R_{q,\eta_2}$$
$$4.u = A^T r + e_1$$
$$5.v = t^T r + e_2 + \text{Decompress}_q(m,1)$$
$$6.\text{return}(c = (\text{Compress}_q(u,d_u), \text{Compress}_q(v,d_v)))$$

Kyber.Dec(sk, c):

$$1.u = \text{Decompress}_q(\text{Compress}_q(u,d_u),d_u)$$
$$2.v = \text{Decompress}_q(\text{Compress}_q(v,d_v),d_v)$$
$$3.m' = \text{Compress}_q(v - s^T u, 1)$$
$$4.\text{return decode}(m')$$

其中 $R_q = \mathbb{Z}_q/(x^n+1)$，$A \xleftarrow{\$} U(R_q^{k\times k})$ 表示 $k \times k$ 维的矩阵 A 中的每一个元素都是多项式，并且是在 R_q 中均匀选取。$s \xleftarrow{\$} R_{q,\eta_1}^k$ 表示 s 系数的范围在 $[-\eta_1, \eta_1]$ 之间的中心二项分布，并且是一个 k 维的列向量。

4.2 Kyber 解密错误概率分析

若不考虑压缩的情况,$m' = e^T r - s^T e_1 + e_2 + m$, 取 $w = e^T r - s^T e_1 + e_2 \in R_q$, 即解密错误概率 $\delta = 1 - P(||w_i|| < \lfloor \frac{q}{4} \rceil)$。其中 $e^T r$ 与 $s^T e_1$ 是两个多项式向量相乘，根据向量相乘法则与 3.2 中多项式相乘法则，计算 $e^T r$ 与 $s^T e_1$ 概率分布共需要进行 $k \times n$ 次卷积操作。

对于压缩函数，可以根据压缩函数的定义遍历所有可能的取值范围，计算压缩后解压缩会带来的错误值与其对应的概率。将对应操作带来的错误分布于压缩函数带来的错误分布进行卷积操作即可得到该操作所产生的错误分布。

5 Saber: 基于 MLWR 方案的错误概率分析

5.1 Saber 设计目标及基本函数

在之前的算法中，都需要进行多次采样操作，并用到拒绝采样这一技术。该过程十分影响算法效率，因此 Saber [4] 依据 LWR 构造，可以使采样次数减半，并且选择模数为 2 的方幂，避免拒绝采样带来的时间损耗。

由于 Saber 中选择的模数都为 2 的方幂，在实现 $\lfloor x \rceil_{q \to p}$ 操作时可以使用函数 $bits(x, \epsilon_q, \epsilon_p) = ((x \gg (\epsilon_q - \epsilon_p)) \& (2^{\epsilon_p} - 1))$，其中 $q = 2^{\epsilon_q}, p = 2^{\epsilon_p}$, 且 $q \geqslant p$。当 $q = p$ 时为模 p 操作。该函数的都是位运算，在具体实现时也有较高的效率。

在给出 Saber 的基本函数之前，先对其中会用到的参数及常量进行简要说明。Saber 中选择三个模数 q, p, T, 并且有 $q = 2^{\epsilon_q}, p = 2^{\epsilon_p}, T = 2^{\epsilon_T}, \epsilon_q > \epsilon_p > \epsilon_T$; 并且用到了三个常量 $h_1 \in R_q, h \in R_q^{l \times 1}, h_2 \in R_q$。其中 h_1 的所有系数都为 $2^{\epsilon_q - \epsilon_p - 1}$; h 中每一个元素都是 h_1; h_2 中所有系数都为 $2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_T - 1} - 2^{\epsilon_q - \epsilon_p - 1}$。

以下为 Saber 中的基本函数。

Saber.KG():

$$1.A \xleftarrow{\$} U(R_q^{l \times l})$$
$$2.s \xleftarrow{\$} R_{q,\mu}^{l \times 1}$$
$$3.b = bits(A^T s + h, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$$
$$4.\text{return } (pk = (A, b), sk = s)$$

Saber.Enc($pk, m \in R_2$):

$$1.s' \xleftarrow{\$} R_{q,\mu}^{l \times 1}$$
$$2.b' = bits(As' + h, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$$
$$3.v' = b^T bits(s', \epsilon_p, \epsilon_p) \in R_p$$
$$4.c_m = bits(v' + h_1 - 2^{\epsilon_p - 1} m, \epsilon_p, \epsilon_T) \in R_T$$
$$5.\text{return } c = (c_m, b')$$

Saber.Dec(sk, c):

$$1.v = b'^T bits(s, \epsilon_p, \epsilon_p) \in R_q$$
$$2.m' = bits(v - 2^{\epsilon_p - \epsilon_T} c_m + h_2, \epsilon_p, \epsilon_p - 1)$$
$$3.\text{return } m'$$

5.2 Saber 解密错误概率分析

在 Saber 中，错误向量实际上是有 $bits()$ 函数引入的，定义错误向量 e, e' 是在舍入是引入的，则有 $b = bits(As + h, \epsilon_q, \epsilon_p) = \frac{p}{q} A^T s + e, b' = \frac{p}{q} As + e'$，该操作与 Kyber 中的压缩函数是类似的，因此可以按照 Kyber 压缩函数中计算错误分布的方法计算 e, e' 的错误分布。

而在对消息进行操作时，也会产生错误向量，设为 e_r, 根据此时 $bits()$ 函数的参数可以确定 $e_r \in [-\frac{p}{2T}, \frac{p}{2T}]$，由于消息的不同会对该分布产生影响，因此为便于分析假设 e_r 中的系数满足均匀分布。

设 $\delta = P(||s'^T e - e'^T s + e_r \bmod p|| > \frac{p}{4})$, 该错误概率的计算方法与 Kyber 类似, 则解密失败的概率为 $1 - \delta$。

6 结语

本文对 NIST 中基于三种不同 LWE 类型 (LWE, RLWE, MLWE) 的密码体制 (Frodo, LAC, Kyber) 的解密错误概率进行了总结。由上述分析可知，解密出现错误的原因根本上来自于对消息的编码与解码方式，如 Frodo 中对消息的编码函数与解码函数与 LP 方案有差异，因此计算解密错误概率时能够接受的范围也与其他体制不同。在计算错误概率时，不同的 LWE 类型其中的运算法则的不同也会导致计算方法有些许差异。

参考文献

[1] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.

[2] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1006–1018, 2016.

[3] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[4] Jan-Pieter D' Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*, pages 282–305. Springer, 2018.

[5] Jan-Pieter D' Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on ring/mod-lwe/lwr based schemes. In *Post-Quantum Cryptography: 10th International Conference, PQCrypto 2019, Chongqing, China, May 8–10, 2019 Revised Selected Papers 10*, pages 103–115. Springer, 2019.

[6] Scott Fluhrer. Cryptanalysis of ring-lwe based key exchange with key share reuse. *Cryptology ePrint Archive*, 2016.

[7] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology–CT-RSA 2011: The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, pages 319–339. Springer, 2011.

[8] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, and Kunpeng Wang. Lac: Practical ring-lwe based public-key encryption with byte-level modulus. *Cryptology ePrint Archive*, 2018.

[9] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

附录: 计算错误概率的 Python 代码

A 基本操作

在 python 代码中，使用字典类型的变量存储某一分布，其他键 (key) 表示该分布能够取道的值，值 (value) 表示对应值的概率。以下函数都定义在 prob_util.py 文件中。

A.1 乘法公式

```
def product_law(A, B):
    """
    Construct the law of the product of independent
        variables from two input laws
    :param A: first input law (dictionary)
    :param B: second input law (dictionary)
    """

    C = {}
    for a in A:
        for b in B:
            c = a*b
            C[c] = C.get(c, 0) + A[a] * B[b]
    return C
```

A.2 字典清理函数

认为当概率小于 2^{-300} 时并不会对后续产生太大影响，将此键值对剔除。

```
def clean_dist(A):
    """
    Clean a distribution to accelerate further computation (drop
        element of the support with proba less than 2^-300)
    :param A: input law (dictionnary)
    """
    B = {}
    for (x, y) in A.items():
        if y>2**(-300):
            B[x] = y
    return B
```

A.3 卷积公式

```
def convolution_law(A, B):
    """
    Construct the convolution of two laws (sum of independent
        variables from two input laws)
    :param A: first input law (dictionary)
    :param B: second input law (dictionary)
    """

    C = {}
    for a in A:
        for b in B:
            c = a + b
            C[c] = C.get(c, 0) + A[a] * B[b]
    return C
```

A.4 自卷积公式

某一分布与自身做 i 次卷积操作。

```
def iter_convolution_law(A, i):
    """
    compute the -ith forld convolution of a distribution (using
        double-and-add)
    :param A: first input law (dictionnary)
    :param i: (integer)
    """
    D = {0: 1.0}
    i_bin = bin(i)[2:] # binary representation of n
    for ch in i_bin:
        D = convolution_law(D, D)
        D = clean_dist(D)
        if ch == '1':
            D = convolution_law(D, A)
            D = clean_dist(D)
    return D
```

A.5 计算总体解密错误概率

```
                def tail_probability(D, t):
                    '''
                    Probability that an drawn from D is strictly greater than t in
                        absolute value
                    :param D: Law (Dictionnary)
                    :param t: tail parameter (integer)
                    '''
                    s = 0
                    ma = abs(max(D.keys()))

                    if t >= ma:
                        return 0
                    for i in reversed(range(int(ceil(t)), ma)): # Summing in
                        reverse for better numerical precision (assuming tails are
                        decreasing)
                        s += D.get(i, 0) + D.get(-i, 0)
                    return s
```

B Frodo 错误概率计算

本部分以 Frodo_1344 为例，描述计算解密错误概率的具体步骤。

B.1 Frodo_1344 概率分布

```
    def frodo1344_dis():
        pro = [18286, 14320, 6876, 2023, 364, 40, 2]
        D = {}
        for i in range(−6, 7):
            D[i] = pro[abs(i)] / (2**16)

        return D
```

B.2 Frodo_1344 解密错误概率计算

```
import prob_util
from math import log

def frodo1344_dis():
    pro = [18286, 14320, 6876, 2023, 364, 40, 2]
    D = {}
    for i in range(−6, 7):
        D[i] = pro[abs(i)] / (2**16)

    return D

class Frodo_1344():
    q = None
    n1 = None
    n2 = None

    R1 = None
    R2 = None
    e1 = None
    e2 = None
    e3 = None

```

```
23                  def ___init___(self, q, n1, n2):
24                      self.q = q
25                      self.n1 = n1
26                      self.n2 = n2
27
28                  def select_distribution(self):
29                      self.R1 = frodo1344_dis()
30                      self.R2 = frodo1344_dis()
31                      self.e1 = frodo1344_dis()
32                      self.e2 = frodo1344_dis()
33                      self.e3 = frodo1344_dis()
34
35                  def failure_prob(self):
36                      self.select_distribution()
37
38                      e2R2 = prob_util.product_law(self.e2, self.R2)
39                      e1R1 = prob_util.product_law(self.e1, self.R1)
40
41                      e2R2 = prob_util.iter_convolution_law(e2R2, self.n2)
42                      e1R1 = prob_util.iter_convolution_law(e1R1, self.n1)
43
44                      D = prob_util.convolution_law(e1R1, e2R2)
45                      F = prob_util.convolution_law(D, self.e3)
46
47                      return prob_util.tail_probability(F, int(self.q / 32))
48
49              if ___name___ == "___main___":
50                  frodo_1344 = Frodo_1344(2**16, 1344, 1344)
51                  f = frodo_1344.failure_prob()
52
53                  print ("failure: %.1f = 2^%.1f"%(f, log(f + 2.**(-300))/log(2)))
```

在 frodo_1344 中 $D = 16, B = 4, q = 2^D$，根据 2.2 节的分析，最终的错误向量小于 2^{D-B-1} 时能够正确解密，与代码 47 行处 $q/32$ 等价。

使用该代码计算出来的错误概率为 $2^{-258.5}$，在 Frodo 提交文档中给出的错误概率为 $2^{-252.5}$.

C LAC 错误概率计算

本部分以 LAC_192 为例，描述计算解密错误概率的具体步骤。

C.1 LAC_192 概率分布

```
def LAC_phi_2():
    D = {1 : 1/8, 0 : 3/4, −1 : 1/8}
    return D
```

C.2 LAC_192 解密错误概率计算

```
import prob_util
from math import log, comb

def LAC_phi_2():
    D = {1 : 1/8, 0 : 3/4, −1 : 1/8}
    return D

class LAC_192():
    n = None
    q = None
    l_t = 8 # can correct 8 errors
    l_n = 256 + 72 # total bit nubmber
    s_Gen = None
    s_Enc = None
    e_Gen = None
    e_Enc = None
    e_Enc_ = None

    def ___init___(self, n, q):
        self.n = n
        self.q = q

    def select_distribution(self):
        self.s_Gen = LAC_phi_2()
        self.s_Enc = LAC_phi_2()
        self.e_Gen = LAC_phi_2()
```

```
27                      self.e_Enc = LAC_phi_2()
28                      self.e_Enc_ = LAC_phi_2()
29
30                  def single_bit_failure_prob(self):
31                      self.select_distribution()
32                      es_ = prob_util.product_law(self.e_Gen, self.s_Enc)
33                      e_s = prob_util.product_law(self.e_Enc, self.s_Gen)
34                      es_ = prob_util.iter_convolution_law(es_, self.n)
35                      e_s = prob_util.iter_convolution_law(e_s, self.n)
36                      D = prob_util.convolution_law(e_s, es_)
37                      F = prob_util.convolution_law(D, self.e_Enc_)
38
39                      return prob_util.tail_probability(F, int(self.q / 4))
40
41                  def dec_failure(self):
42                      bit_failure = self.single_bit_failure_prob()
43                      bit_failure_ = 1 - bit_failure
44                      pro = 0.0
45                      for i in range(self.l_t + 1, self.l_n + 1):
46                          pro += comb(self.l_n, i) * (bit_failure**i) * (bit_failure_
                                ** (self.l_n - i))
47
48                      return pro
49
50              if __name__ == "__main__":
51                  lac_192 = LAC_192(1024, 251)
52
53                  f = lac_192.single_bit_failure_prob()
54                  print ("failure: %.2f = 2^%.2f"%(f, log(f + 2.**(-300))/log(2)))
55
56                  dec_f = lac_192.dec_failure()
57                  print ("failure: %.2f = 2^%.2f"%(f, log(dec_f + 2.**(-300))/log(2)
                        ))
```

使用该代码计算出来的单个比特错误概率为 $2^{-23.81}$，整体解密错误概率为 2^{157}。在 LAC 提交文档中单个比特错误概率为 $2^{-22.27}$，整体解密错误概率为 2^{-143}。

在上述代码中，仍有部分步骤的没有考虑到，如将 n 次多项式转化为 l_v 次多项式。这导致计算出来的概率偏小，但仍在可接受范围内。

D Kyber 错误概率计算

本部分以 Kyber_768 为例，描述计算解密错误概率的具体步骤。

D.1 中心二项分布

```
def centered_binomial_distribution(eta):
    """
    Probability density function of the centered binomial distribution for
        −eta to eta
    :param eta: integer
    :returns: A dictionary {x:p(x) for x in {−eta .. eta}}
    """
    return {k: comb(2∗eta, eta +k) ∗ (0.5)∗∗(2∗eta) for k in range(−
        eta, eta+1)}
```

D.2 Kyber 压缩函数中的错误概率

```
def mod_switch(x, q, rq):
    """ Modulus switching (rounding to a different discretization of the
        Torus)
    :param x: value to round (integer)
    :param q: input modulus (integer)
    :param rq: output modulus (integer)
    """
    return int(round(1.∗ rq ∗ x / q) % rq)
```

```
def mod_centered(x, q):
    """ reduction mod q, centered (ie represented in −q/2 .. q/2)
    :param x: value to round (integer)
    :param q: input modulus (integer)
    """
    a = x % q
    if a < q/2:
        return a
    return a − q
```

```
def build_mod_switching_error_law(q, rq):
    """ Construct Error law: law of the difference introduced by
        switching from and back a uniform value mod q
    :param q: original modulus (integer)
    :param rq: intermediate modulus (integer)
    """
    D = {}
    V = {}
    for x in range(q):
        y = mod_switch(x, q, rq)
        z = mod_switch(y, rq, q)
        d = mod_centered(x − z, q)
        D[d] = D.get(d, 0) + 1./q
        V[y] = V.get(y, 0) + 1

    return D
```

D.3 Kyber_768 解密错误概率计算

```
import prob_util
from math import log, comb

class Kyber_768():
    q = 3329
    n = 256
    k = 3
    eta_1 = 2
    eta_2 = 2
    d_u = 10
    d_v = 4

    e = None
    r = None
    s = None
    e_1 = None
    e_2 = None
    compress_u = None
    compress_v = None
    compress_t = None

    def ___init___(self):
        self.e = prob_util.centered_binomial_distribution(self.eta_1)
        self.r = prob_util.centered_binomial_distribution(self.eta_1)
        self.s = prob_util.centered_binomial_distribution(self.eta_1)
        self.e_1 = prob_util.centered_binomial_distribution(self.eta_2
            )
        self.e_2 = prob_util.centered_binomial_distribution(self.eta_2
            )
        self.compress_u = build_mod_switching_error_law(self.q,
            2**10)
        self.compress_v = build_mod_switching_error_law(self.q,
            2**4)
        self.compress_t = build_mod_switching_error_law(self.q,
            2**12)

    def failure_prob(self):
```

```
33                  compress_s = prob_util.convolution_law(self.s, self.compress_t
                        )
34                  compress_s = prob_util.product_law(compress_s, self.e_1)
35                  B1 = prob_util.product_law(self.r, compress_s)
36                  C1 = prob_util.iter_convolution_law(B1, self.n * self.k)
37
38                  compress_e = prob_util.convolution_law(self.e, self.
                        compress_u)
39                  B2 = prob_util.product_law(self.s, compress_e)
40                  C2 = prob_util.iter_convolution_law(B2, self.n * self.k)
41
42                  C = prob_util.convolution_law(C1, C2)
43                  F = prob_util.convolution_law(self.compress_v, self.e_2)
44                  D = prob_util.convolution_law(C, F)
45                  return prob_util.tail_probability(D, self.q / 4)
46
47          if __name__ == "__main__":
48              kyber_768 = Kyber_768()
49              f = kyber_768.failure_prob()
50              print ("failure: %.1f = 2^%.1f"%(f, log(f + 2.**(-300))/log(2)))
```

使用上述代码计算的解密错误概率为 $2^{-169.7}$, 提交文档中的解密错误概率为 2^{-164}, 误差可以接受。

E Saber 错误概率计算

```
import prob_util
from math import log, comb

def mod_switch(x, q, rq):
    """ Modulus switching (rounding to a different discretization of the
        Torus)
    :param x: value to round (integer)
    :param q: input modulus (integer)
    :param rq: output modulus (integer)
    """
    return int(round(1.* rq * x / q) % rq)


def mod_centered(x, q):
    """ reduction mod q, centered (ie represented in −q/2 .. q/2)
    :param x: value to round (integer)
    :param q: input modulus (integer)
    """
    a = x % q
    if a < q/2:
        return a
    return a − q

def build_mod_switching_error_law(q, rq):
    """ Construct Error law: law of the difference introduced by switching
        from and back a uniform value mod q
    :param q: original modulus (integer)
    :param rq: intermediate modulus (integer)
    """
    D = {}
    V = {}
    for x in range(q):
        y = mod_switch(x, q, rq)
        z = mod_switch(y, rq, q)
        d = mod_centered(x − z, q)
        D[d] = D.get(d, 0) + 1./q
        V[y] = V.get(y, 0) + 1
```

```
36
37              return D
38
39          def centered_binomial_distribution(eta):
40              """
41              Probability density function of the centered binomial distribution for −
                    eta to eta
42              :param eta: integer
43              :returns: A dictionary {x:p(x) for x in {−eta .. eta}}
44              """
45              return {k: comb(2∗eta, eta +k) ∗ (0.5)∗∗(2∗eta) for k in range(−eta,
                    eta+1)}
46
47          def convolution_remove_dependency(A, B, q, p):
48              normalizer = {}
49              maxa = q / p
50              for a in A:
51                  normalizer[a % maxa] = normalizer.get(a % maxa, 0) + A[a]
52
53              C = {}
54              for a in A:
55                  for b in B:
56                      c = a − b
57                      if (c % maxa == 0):
58                          C[c] = C.get(c, 0) + A[a] ∗ B[b] / normalizer[a % maxa]
59
60              return C
61
62          class Saber():
63              l = 3
64              n = 256
65              q = 2∗∗13
66              p = 2∗∗10
67              T = 2∗∗4
68              mu = 8 // 2
69
70              def failure_prob(self):
71                  s = prob_util.centered_binomial_distribution(self.mu) # pdf secret
72                  e = build_mod_switching_error_law(self.q, self.p)  # pdf error
```

```
                se = prob_util.product_law(s, e)
                se2 = prob_util.iter_convolution_law(se, self.n * self.l)
                se2 = convolution_remove_dependency(se2, se2, self.q, self.p)

                e2 = build_mod_switching_error_law(self.q, self.T)

                D = prob_util.convolution_law(se2, e2)

                pro = prob_util.tail_probability(D, self.q/4.)
                return pro

        if __name__ == "__main__":
            saber = Saber()
            f = saber.failure_prob()
            print ("failure: %.1f = 2^%.1f"%(f, log(f + 2.**(-300))/log(2)))
```

使用该代码计算出来的错误概率为 2^{-144}，第三轮提交文档中给出的错误概率为 2^{-136}，误差在可接受范围内。