环上舍入学习和模上舍入学习的通用实现算法与参数选取方法

姜子铭[1),2)] 周永彬[1),2),3)] 张 锐[1),2)]

[1)](中国科学院信息工程研究所 北京 100093)

[2)](中国科学院大学网络空间安全学院 北京 100049)

[3)](南京理工大学网络空间安全学院 南京 210094)

摘 要 格密码领域中的环上舍入学习(RLWR)和模上舍入学习(MLWR)问题是构造后量子密码原语的一类重要数学工具,已广泛应用于伪随机函数、陷门函数等基础密码构造. RLWR 和 MLWR 实现通常包括三项基础操作:多项式乘法、模约化和舍入计算,三项基础操作均有多种适用于不同参数的实现方案,相同运行平台、相同安全等级下 RLWR 和 MLWR 软件实现效率受参数影响显著. 然而,现有 RLWR 和 MLWR 方案实现及效率优化工作大多仅针对某些特定参数,无法处理任意参数;此外,现有 RLWR 和 MLWR 方案参数选取大多只考虑了部分基础操作的效率,缺乏系统的快速实现参数选取方法. 为解决上述问题,本文提出了通用高效的 RLWR 实现算法和 MLWR 实现算法,以及 RLWR 和 MLWR 快速实现参数选取方法. 本文首先给出了 NTT 和 NTT 负折叠卷积的使用条件与方案参数以及 CPU 字长之间的量化关系,扩展了可快速实现的基于多项式环的密码方案参数空间;其次,提出了一种适用于 RLWR 和 MLWR 实现的新舍入算法,与通用的传统舍入实现相比,新舍入算法的效率在 64 位 Intel i7 平台下提高了 11%左右;最后,提出了通用 RLWR 实现算法和 MLWR 实现算法,通用实现算法根据方案参数和 CPU 字长灵活选取高效的基础操作实现方案,将其应用于 Saber 方案实现,在 64 位 Intel i7 平台下未使用编译优化指令的 Saber 密钥封装效率提升了 52%左右. 此外,对比分析了不同参数下 RLWR 和 MLWR 的效率,提出了 RLWR 和 MLWR 快速实现参数选取方法,为 RLWR 和 MLWR 方案设计与实现中的参数选取提供了指导.

关键词 格密码;舍入学习;多项式乘法;数论变换;舍入计算

中图法分类号 TP309 **DOI 号** 10.11897/SP.J.1016.2022.01326

The General Implementation Algorithms and the Parameters Selection Methods of Ring Learning with Rounding and Module Learning with Rounding

JIANG Zi-Ming[1),2)] ZHOU Yong-Bin[1),2),3)] ZHANG Rui[1),2)]

[1)](*Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS), Beijing* 100093)

[2)](*School of Cyber Security, University of Chinese Academy of Sciences (UCAS), Beijing* 100049)

[3)](*School of Cyber Security, Nanjing University of Science and Technology (NJUST), Nanjing* 210094)

Abstract The learning with rounding (LWR) problem in lattice cryptography is one of the fundamental tools for constructing post-quantum cryptographic primitives. As a deterministic variant of learning with errors (LWE), LWR replaces random errors in LWE with deterministic rounding. The removal of random error sampling makes LWR-based cryptosystems more efficient than LWE. Since it was proposed, LWR problem has been extensively applied to the construction of basic cryptosystems, such as low-depth pseudorandom functions, lossy trapdoor functions and public-key encryption schemes. In order to reduce the computational complexity and bandwidth,

收稿日期:2021-06-03;在线发布日期:2021-11-18. 本课题得到国家自然科学基金(61632020,U1936209,62002353)、北京市自然科学基金(4192067)资助. **姜子铭**,博士研究生,主要研究方向为格密码. E-mail: jiangziming@iie.ac.cn. **周永彬**(通信作者),博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为网络与信息安全理论及技术. E-mail: zhouyongbin@iie.ac.cn. **张 锐**,博士,研究员,博士生导师,中国计算机学会(CCF)会员,主要研究领域为信息安全、密码学、密码工程学.

the ring and module version of LWR, that is, ring LWR (RLWR) and module LWR (MLWR), are mostly used in practice. To make the post-quantum cryptosystems more practical, it is of great significance to improve the efficiency of RLWR and MLWR. The ring in lattice usually takes the polynomial ring, so RLWR/MLWR includes three basic operations: polynomial multiplication, modular reduction, and rounding. For each of the three operations, there are a variety of implementation methods with different efficiencies which apply to different parameters. The efficiency of RLWR/MLWR with different parameters may vary greatly at the same security level. However, most of the existing implementations of RLWR and MLWR are only applicable to some certain schemes with specific set of parameters. In addition, most of the existing RLWR and MLWR schemes have not given consideration to the implementation efficiency of all basic operations. There is a lack of RLWR/MLWR parameter selection method for the purpose of achieving high efficiency. In order to solve the above problems, we propose the general and efficient implementation algorithms of RLWR and MLWR, as well as the parameters selection methods of RLWR and MLWR to achieve high efficiency. In this work, we first discuss the conditions of existing implementation methods of the three basic operations on CPU platform and focus on the special parameters commonly used in lattice-based cryptosystems. In particular, we clarify the conditions of NTT-based multiplication and negative wrapped convolution, so we can also use NTT to accelerate polynomial multiplication when the modulus of RLWR/MLWR does not meet the requirements of NTT modulus but meets the conditions in this work. As a result, the parameter space of high-efficient cryptosystems based on polynomial ring structures would be expanded. Secondly, we present a new rounding algorithm that uses shift, addition and subtraction operations based on precomputation instead of the division operation. The efficiency of the new rounding algorithm is improved by about 11% compared with the universal traditional rounding algorithm which precomputes the reciprocal of the RLWR/MLWR modulus. Although the efficiency of the new rounding algorithm is about 2.5% lower than the most efficient rounding method based on addition and shift operations, the new rounding algorithm may be applicable when the addition-then-shift operation cannot be used, especially when the modulus of RLWR/MLWR is NTT modulus. Finally, we propose general implementation algorithms of RLWR and MLWR. Their core idea is to select efficient implementation schemes of the three basic operations according to the parameters and the CPU word length. When it is applied to Saber scheme, the efficiency of Saber key encapsulation is improved about 52% on 64-bit Intel i7 platform without compiler optimization instruction. We then compare the efficiency of RLWR/MLWR with different parameters and propose parameters selection methods of RLWR and MLWR. It aims to provide readers with reference for parameter selection in the design and implementation of RLWR/MLWR-based cryptosystems.

Keywords　lattice-based cryptography; learning with rounding; polynomial multiplication; number theory transformation; rounding

1　引　言

一旦通用量子计算机问世，现有广泛使用的基于大数分解和离散对数等问题的传统公钥密码体制将被攻破[1]，发展能抵抗量子攻击的后量子密码体制是当前的研究热点，也是迫切的现实需求. 美国国家标准与技术研究院(National Institute of Standards and Technology，NIST)和中国密码学会先后举办了后量子密码算法征集活动. 目前公开的后量子公钥密码主要包括：基于格的密码体制、基于编码的密码体制、基于多变量的密码体制和基于哈希的密码

体制，其中，基于格的密码体制凭借其抵抗已知量子攻击、可并行实现以及存在“最坏情形”到“平均情形”的底层困难问题安全性归约等优势，被认为是最有前景的后量子密码候选之一.

舍入学习(Learning with Rounding，LWR)作为格密码中应用最广泛的确定误差底层函数，是后量子密码原语的重要构造工具，可直接用于构造伪随机函数和陷门函数等[2-3]. 为构造基于格的伪随机函数，Banerjee 等人[2]于 2012 年提出 LWR 问题，并给出超多项式级别模数下带错学习(Learning with Errors，LWE)到 LWR 的归约. 随后，Alwen 等人[3]借助有损采样完成了多项式级别模数下的归约，但模数 p 和 q 仍需满足 $Q\geq 2nmEp$ 且 Q^2 不整除 q，其中 Q 是 q 的最大素因子，n 为向量维数，m 为采样个数，E 为 LWE 的误差边界. Bogdanov 等人[4]进一步改进了归约方法，解除了多项式级别下对模数的其它限制条件，模数只需满足 $q\geq 2mEp$.

LWR 可视为去随机化的 LWE，由于不再需要随机误差采样，LWR 的实现效率更高、抗侧信道攻击能力更强，已广泛应用于低深度伪随机函数[5]、有损陷门函数[3]、确定性公钥加密[6-9]以及全同态加密[10]等基础密码构造. 实际应用中需考虑计算复杂度和带宽等效率指标，现有实用的格密码方案大多采用环结构或模结构，LWR 的环版本和模版本分别称为环上舍入学习(Ring LWR，RLWR)和模上舍入学习(Module LWR，MLWR). NIST 后量子密码标准算法征集活动中的 Lizard[7]提案和 Round5[8]提案提供了基于 RLWR 的公钥加密方案，Saber[9]方案是基于 MLWR 的公钥加密方案，特别地，Saber 方案入围了公钥加密的决赛. 考虑到后量子密码体制实用化的现实需求，提升 RLWR 和 MLWR 实现效率具有重要意义.

RLWR 分布 $(a,b=\lfloor a\cdot s\rceil_p)$ 由参数 N,f,p,q 和环元素 s 决定，其中 N,p,q 是正整数，f 是单变元 N 次首一多项式，$a,b,s\in R_q=\mathbb{Z}_q[x]/(f)$. 考虑到 s 通常取自 0-1 分布等特殊分布，方案实现中 s 系数的上界可能远小于 q，本文假设多项式 s 系数绝对值的上界为 $\frac{1}{2}S$. RLWR 分布中 $b(x)$ 的计算分为三步：

① 整数域上的多项式乘法，$b_1(x)=a(x)\cdot s(x)$；

② 模约化，$b_2(x)=((b_1(x) \bmod f(x)) \bmod q)$；

③ 舍入计算，$b(x)=\left\lfloor \frac{p}{q}\cdot b_2(x)\right\rceil$.

MLWR 分布 $(\boldsymbol{a},\boldsymbol{b}=\lfloor(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{s})\rceil_p)$ 由参数 N,f,p,q,l 和向量 $\boldsymbol{s}$ 决定，其中 l 是正整数，其他参数与 RLWR 相同，$\boldsymbol{a}\in R_q^{l\times l}$，$\boldsymbol{b},\boldsymbol{s}\in R_q^l$，本文假设方案实现中向量 $\boldsymbol{s}$ 所有元素的系数绝对值的上界为 $\frac{1}{2}S$. MLWR 分布中向量 $\boldsymbol{b}$ 的计算同样分为三步，与 RLWR 的区别在于，步骤①中计算的是矩阵乘法，等价于 $l\times l$ 次整数域上的多项式乘法和 $l\times(l-1)$ 次整数域上的多项式加法. 现有 RLWR 和 MLWR 实现通常包括三项基础操作：多项式乘法、模约化和舍入计算，三者均有多种实现方案，RLWR 以及 MLWR 的实现即为三项基础操作实现方案的组合.

多项式乘法是格密码领域最常用的基础运算之一，在基于 NTRU、RLWE 和 RLWR 等假设的密码体制中，计算两个多项式的乘积是一个基本操作，往往也是耗时最高的操作. 常用多项式乘法快速实现算法包括 Karatsuba[11]、Toom-Cook[12-13]、快速傅里叶变换[14](Fast Fourier Transform，FFT)、数论变换[15](Number Theoretic Transform，NTT)以及稀疏乘法算法，格密码领域多项式乘法的实现算法往往取决于方案参数[16]：模数 q 是 NTT 模数时通常采用 NTT 乘法算法，例如 Kyber[17]、Dilithium[18]和 qTESLA[19]方案；对于模数不是 NTT 模数的方案，次数界 N 较小时通常采用 Karatsuba 或 Toom-Cook 算法，例如 Saber 方案，N 较大时通常采用 FFT 乘法算法，例如 LIMA[20]方案；此外，s 的系数取自集合 $\{-1,0,1\}$ 时通常采用稀疏乘法算法，例如 Round5 和 Lizard 方案. 上述多项式乘法实现算法效率差异显著，其中 NTT 乘法算法的理论复杂度最低且实际效率相对较高. 虽然很多格密码方案实现采用了 NTT 乘法算法，但这些高效的 NTT 实现大多针对特定参数，无法处理任意参数.

此外，现有格密码方案大多认为，方案模数 q 是 NTT 模数时才可使用 NTT 加速多项式乘法[9]. 然而，NTT 乘法算法理论上没有限制条件，实用中仅对 NTT 模数的大小有要求，这是因为计算机能直接处理的整数存在上下界，NTT 模数过大时，引入大整数运算反而会降低效率. 事实上，即使 q 不满足 NTT 模数的一般性要求，若整数域上乘积多项式的系数较小，即参数 N,q,S 较小，使得存在 NTT 模数 M 满足 $qSN<2M$，也可以采用 NTT 乘法算法. 例如采用 NTT 乘法的 Saber 实现[21]，其密钥封装效率在 Cortex-M4 平台与原始实现相比提高了 22%左右. 目前，格密码领域对 NTT 实际使用条件的研究工作较少，现有计算平台下 NTT 的限制条

件与格密码参数之间的关系尚待研究.

格密码领域通常涉及两种模约化操作：模多项式和模整数. 格密码方案通常取 $R_q=\mathbb{Z}_q[x]/(x^N+1)$，模多项式采用减法实现. 模整数运算主要的计算开销是除法，常用通用模整数快速实现算法包括 Montgomery 算法[22] 和 Barrett 算法[23]，二者都是在预计算的基础上，利用加减、乘法和移位代替除法. 现有格密码方案模数 q 通常取特殊值，采用特殊模整数实现方法：q 为 2 的方幂时采用按位与操作，例如 Round5、Saber 和 Lizard 方案；q 是 NTT 模数时，采用模数为 q 的 NTT 乘法算法，可省略步骤② 模约化中的模 q 运算，NTT 算法内部模约化通常采用 Montgomery 算法，例如 Kyber、Dilithium 和 qTESLA 方案. 现有计算平台下，Barrett 算法和 Montgomery 算法是否是格密码领域效率最高的通用模整数算法尚待研究.

舍入计算同样是格密码领域的基础运算，除计算 LWR 分布外，还应用于压缩与解压缩、编码与解码等模块. 与模整数类似，舍入计算主要的计算开销是除法. 目前，RLWR/MLWR 参数 q 和 p 通常取 2 的方幂，显然，若模数比值 q/p 是 2 的方幂，则可使用先加法再移位操作计算舍入值，例如 Round5、Saber 和 Lizard 方案. 模数比值 q/p 非 2 的方幂的方案较少，通用舍入计算优化方法尚待研究.

综上所述，RLWR 以及 MLWR 的实现即为三项基础操作实现方案的组合，三者均有多种实现方案，效率较高的实现方案都只适用于特殊参数：NTT 乘法实现大多要求模数 q 为 NTT 模数（通常取奇素数），采用按位与实现模整数、先加法再移位实现舍入计算均要求模数 q 的因子包含 2 的方幂. 故参数 q 不能同时满足上述限制条件. 为提升模约化、舍入计算和随机数采样的效率，RLWR 和 MLWR 的参数 q 和 p 通常取 2 的方幂，而现有 NTT 实现大多无法处理这种模数. 目前，RLWR 以及 MLWR 实现主要存在以下两个问题：

(1) 现有 RLWR 和 MLWR 实现难以兼顾通用性与高效性. 虽然已有许多 RLWR 和 MLWR 实现及优化工作[7-9]，但这些实现方法大多针对特定密码方案，只适用于某些特殊参数，无法处理任意参数.

(2) 缺乏系统的 RLWR 和 MLWR 快速实现参数选取方法. 多项式乘法是三项基础操作中耗时最高的，但现有 RLWR 和 MLWR 方案选取参数时大多未考虑多项式乘法的效率，缺乏系统性研究分析.

针对以上问题，本文提出通用的 RLWR 实现算法和 MLWR 实现算法，以及 RLWR 和 MLWR 参数选取方法，具体贡献如下：

(1) 扩展现有可采用 NTT 类乘法算法的格密码方案参数空间. 本文给出通用 CPU 平台下，NTT 和 NTT 负折叠卷积的限制条件与格密码方案参数 N，f，q，S 和 CPU 字长 B 之间的量化关系. 当参数满足本文给出的限制条件时，即使模数 q 不满足现有 NTT 类乘法实现的一般性要求，也可以采用 NTT 类乘法算法. 除 RLWR/MLWR 实现外，该成果可广泛应用其它于基于多项式环的密码方案实现，例如，基于 Ring-LWE 和 NTRU 等假设的密码方案.

(2) 提出一种适用于 RLWR 和 MLWR 实现的新舍入算法. 当 q 是奇素数且 q 可作为 NTT 模数时，舍入计算无法采用最高效的先加法再移位操作. 针对这类 q，本文提出一种基于预计算的新舍入算法，在 64 位通用 CPU 平台，其效率与传统通用舍入实现相比提高 11%左右，虽然该算法的效率略低于先加法再移位，但二者的效率相差不超过 2.5%. 新舍入算法适合搭配 NTT 乘法算法使用，此外还可以应用于压缩与解压缩、编码与解码等其他密码学常用模块的实现.

(3) 提出通用 RLWR 实现算法和 MLWR 实现算法. 本文根据方案参数和 CPU 字长，灵活选择三项基础操作实现方案的优化组合方式，在保证通用性的情况下达到较高的软件实现效率. 将该成果应用于 NIST 后量子密码算法征集中的 Saber 方案，与 Saber 第三轮原始实现相比，未使用编译优化指令时 Saber 密钥封装的效率可提升 52% 左右.

(4) 进行实验对比分析，提出 RLWR 和 MLWR 快速实现参数选取方法. 本文完成了相关算法的 C 语言实现，在 64 位 Intel(R) Core(TM) i7-6700 CPU@3.40 GHz，ubuntu16.04 操作系统上使用 GCC 编译器测试了各算法的实际运行时间. 根据实际效率，本文提出 RLWR 和 MLWR 快速实现参数选取方法. 在参数选择合理的情况下，512 次多项式 a 和 s 系数均取自均匀分布的 RLWR 的运行时间最快为 45 μs 左右，基本满足实用预期.

2　预备知识

2.1　符号说明

$\mathbb{Z}$、$\mathbb{N}$：整数集合、自然数集合；

$\mathbb{Z}_+$：正整数集合；

$\mathbb{Z}_q$：对 $\forall q\in\mathbb{Z}_+$，$\mathbb{Z}_q$ 表示 $\mathbb{Z}$ 模 q 的剩余类，本文默认取绝对最小剩余系 $\left\{-\left\lfloor\frac{q-1}{2}\right\rfloor,\cdots,0,\cdots,\left\lfloor\frac{q}{2}\right\rfloor\right\}$；

$\mathbb{Z}[x]$：记 x 是一个变元，$\mathbb{Z}[x]$ 表示所有整系数多项式构成的集合；

$\mathbb{Z}_q[x]$：记 x 是一个变元，对 $\forall q\in\mathbb{Z}_+$，$\mathbb{Z}_q[x]$ 表示所有系数属于 $\mathbb{Z}_q$ 的多项式构成的集合；

$R=\mathbb{Z}[x]/(f)$：对 $\forall f\in\mathbb{Z}[x]$，$(f)$ 表示由 f 生成的理想，$\mathbb{Z}[x]/(f)$ 表示 $\mathbb{Z}[x]$ 模 (f) 剩余类环，记作 R；

$R_q=\mathbb{Z}_q[x]/(f)$：对 $\forall q\in\mathbb{Z}_+$，$\mathbb{Z}_q[x]/(f)$ 表示 $\mathbb{Z}_q[x]$ 模 (f) 剩余类环，记作 R_q；

a,A：白体字母表示集合中的元素；

$\boldsymbol{a},\boldsymbol{A}$：黑体字母表示向量或矩阵；

$\boldsymbol{a}^{\mathrm{T}}$：向量或矩阵的转置；

log：如不做特殊说明，本文中对数默认以 2 为底；

$\lfloor x\rceil$、$\lceil x\rceil$、$\lfloor x\rceil$：向下取整、向上取整、舍入取整.

$\gcd(a,b)$：a 和 b 最大公因子.

2.2 LWR、RLWR 和 MLWR

对任意整数 $q\geq p\geq 2$，舍入函数 $\lfloor\cdot\rceil_p:\mathbb{Z}_q\to\mathbb{Z}_p$ 定义为 $\lfloor x\rceil_p=\left\lfloor\left(\frac{p}{q}\right)\cdot x\right\rceil \bmod p$.

定义 1. LWR 分布. 假设 $N,p,q,m\in\mathbb{Z}_+$，其中，$q\geq p\geq 2$. 给定向量 $\boldsymbol{s}\in\mathbb{Z}_q^{N\times 1}$，LWR 分布 L_{LWR} 定义为 $\mathbb{Z}_q^{N\times m}\times\mathbb{Z}_p^m$ 上的分布 $(\boldsymbol{a},\boldsymbol{b})$，

$$L_{\mathrm{LWR}}=\{(\boldsymbol{a},\boldsymbol{b})\mid\boldsymbol{a}\leftarrow\mathbb{Z}_q^{N\times m},\boldsymbol{b}=\lfloor(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{s})\rceil_p\in\mathbb{Z}_p^m\},$$

其中 $\boldsymbol{a}\leftarrow\mathbb{Z}_q^{N\times m}$，表示矩阵 $\boldsymbol{a}$ 的每一个元素都取自 $\mathbb{Z}_q$ 上的均匀随机分布.

定义 2. RLWR 分布. 假设 $N,p,q\in\mathbb{Z}_+$，其中，$q\geq p\geq 2$；$f(x)=\sum_{i=0}^{N}f_ix^i$ 是 N 次多项式；环 $R_q=\mathbb{Z}_q[x]/(f)$. 给定环元素 $s\in R_q$，RLWR 分布 L_{RLWR} 定义为 $R_q\times R_p$ 上的分布 (a,b)，

$$L_{\mathrm{RLWR}}=\{(a,b)\mid a\leftarrow R_q,b=\lfloor a\cdot s\rceil_p\in R_p\},$$

其中 $a\leftarrow R_q$，表示环元素 a 取自环 R_q 上的均匀随机分布，多项式运算在环 R_q 中进行.

定义 3. MLWR 分布. 假设 $N,p,q,l,m\in\mathbb{Z}_+$，其中，$q\geq p\geq 2$；$f(x)=\sum_{i=0}^{N}f_ix^i$ 是 N 次多项式；环 $R_q=\mathbb{Z}_q[x]/(f)$. 给定环元素 $s\in R_q^{l\times 1}$，MLWR 分布 L_{MLWR} 定义为 $R_q^{l\times m}\times R_p^m$ 上的分布 $(\boldsymbol{a},\boldsymbol{b})$，

$$L_{\mathrm{MLWR}}=\{(\boldsymbol{a},\boldsymbol{b})\mid\boldsymbol{a}\leftarrow R_q^{l\times m},\boldsymbol{b}=\lfloor(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{s})\rceil_p\in R_p^m\},$$

其中 $\boldsymbol{a}\leftarrow R_q^{l\times m}$ 表示矩阵 $\boldsymbol{a}$ 的每一个元素都取自环 R_q 上的均匀随机分布，多项式运算在环 R_q 中进行.

RLWR 分布由参数 N,f,p,q 和环元素 s 决定. 格密码中多项式环通常取 $R_q=\mathbb{Z}_q[x]/(x^N+1)$. a 取自 R_q 上的均匀随机分布，记 $a=\sum_{i=0}^{N-1}a_ix^i$，其系数 $|a_i|\leq\frac{1}{2}q$. s 除均匀分布外，还可能服从 0-1 分布、高斯分布等特殊分布，s 系数实际的上界可能远小于 q，本文假设方案实现中 s 系数绝对值的上界为 $\frac{1}{2}S$，记 $s=\sum_{i=0}^{N-1}s_ix^i$，其系数 $|s_i|\leq\frac{1}{2}S$，其中正整数 $S\leq q$；此外，若 s 系数等于 0 的概率较大，则可采用某些特殊多项式乘法算法，本文用参数 $\mathrm{P}[s_i\neq 0]$ 表示 s 的系数不为 0 的概率. 给定 s 的分布后即可确定参数 S 和 $\mathrm{P}[s_i\neq 0]$ 的值. RLWR 的实现方案取决于参数 N,f,p,q,S 和 $\mathrm{P}[s_i\neq 0]$.

MLWR 方案大多取 $m=l$，此时 MLWR 分布由参数 N,f,p,q,l 和向量 $\boldsymbol{s}$ 决定. MLWR 参数 $l=m=1$ 时即为 RLWR，参数 N,f,p,q 含义与 RLWR 相同. 向量 $\boldsymbol{s}$ 中的元素均取自环 R_q 上的某一分布，给定该分布后即可确定参数 S 和 $\mathrm{P}[s_i\neq 0]$ 的值. MLWR 的实现方案取决于参数 N,f,p,q,l,S 和 $\mathrm{P}[s_i\neq 0]$.

2.3 FFT 和 NTT

多项式有两种表示方法：系数表达和点值表达. 假设 $a(x)$ 的次数是 k，任意一个大于 k 的整数都是该多项式的次数界. 对于次数界为 N 的多项式，系数表达即 $a(x)=\sum_{j=0}^{N-1}a_jx^j$，记 $\boldsymbol{a}=(a_0,\cdots,a_{N-1})$ 为系数向量；点值表达是由 N 个点值对组成的集合 $\{(x_i,y_i=a(x_i))\}_{0\leq i<N}$. 若 $c(x)=a(x)\cdot b(x)$，那么 $c(x)$ 对应的点值表达就是 $\{(x_i,a(x_i)\cdot b(x_i))\}_{0\leq i<2N}$. 系数表达的多项式乘法经典算法复杂度为 $O(N^2)$，而点值表达的多项式乘法复杂度为 $O(N)$. 通过 FFT 或 NTT 对多项式的表示方法进行转换，可将系数表达的多项式乘法复杂度降低为 $\Omega(N\log N)$，见图 1[24]，注意，乘积的次数界为 $2N$，因此需加倍次数界.

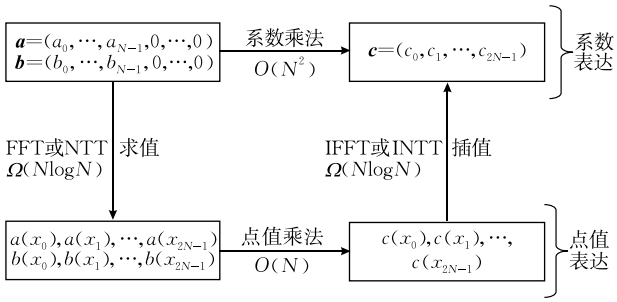

图 1 多项式的系数乘法与点值乘法

作为分治算法，FFT 和 NTT 的输入规模 n 通常取 2 的方幂，假设输入多项式次数界为 N，一般取 $n=2^{\lceil \log N \rceil+1}$. FFT 选择 n 次单位复数根 $\omega=\mathrm{e}^{2k\pi \mathrm{i}/n}$，$k=0,1,\cdots,n-1$ 作为求值点，其中 i 为虚数单位，这些点被称为旋转因子；$\omega_n=\mathrm{e}^{2\pi \mathrm{i}/n}$ 称为主 n 次单位复数根，其他单位根都是 ω_n 的方幂. 定义两个次数界为 $n/2$ 的多项式

$$a^{[0]}(x)=a_0+a_2x+a_4x^2+\cdots+a_{n-2}x^{n/2-1},$$

$$a^{[1]}(x)=a_1+a_3x+a_5x^2+\cdots+a_{n-1}x^{n/2-1},$$

于是有 $a(x)=a^{[0]}(x^2)+x\cdot a^{[1]}(x^2)$. FFT 的基本思想是分治，也就是将对 $a(x)$ 的求值分成对 $a^{[0]}(x^2)$ 和对 $a^{[1]}(x^2)$ 的求值，求值点为 $(\omega_n^k)^2$，$k=0,1,\cdots,n-1$. 由于 $\omega_n^0,\cdots,\omega_n^{n-1}$ 在乘法的意义下形成一个群，故 $(\omega_n^0)^2,\cdots,(\omega_n^{n-1})^2$ 仅由 $n/2$ 个 $n/2$ 次单位复数根组成，因此分治后算法的输入规模减半. NTT 与 FFT 算法的原理基本相同，不同之处在于，NTT 选择有限群 $(\mathbb{Z}_M,\times)$ 的原根 g 及其方幂作为求值点. 简单地说，任意素数 M 及其原根 g 满足 $g^{M-1}\equiv 1 \pmod M$，若 n 恰好整除 $M-1$，那么令 $k=(M-1)/n$，取 $\alpha=g^k$，则可以将 α 视为主 n 次单位.

FFT 乘法算法（FFT-Polynomial Multiplication Algorithm，FFT-PMA）见算法 1，NTT 乘法算法（NTT-PMA）见算法 2，其中 FFT 算法和 NTT 算法见附录 A. 将多项式系数按调用顺序重新排列即可实现迭代 FFT/NTT，能节省少量运行时间. 特别地，输入规模确定后，单位根等变量的值是固定的，因此可通过预计算旋转因子等值进一步提高效率.

算法 1.　FFT-PMA($\boldsymbol{a},\boldsymbol{b},N$).

输入：$\boldsymbol{a},\boldsymbol{b},N$. 其中 $\boldsymbol{a},\boldsymbol{b}$ 是系数向量，N 是次数界

输出：$c(x)$ 的系数向量 $\boldsymbol{c}$，满足 $c(x)=a(x)\cdot b(x)$

1. 加倍次数界. 在向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的高位添加值为 0 的元素，直至向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的长度为 $n=2^{\lceil \log N \rceil+1}$.
2. 求值. 调用输入规模为 n 的 FFT 算法，计算 $\boldsymbol{a}$ 对应的点值向量 $\boldsymbol{A}$，和 $\boldsymbol{b}$ 对应的点值向量 $\boldsymbol{B}$.
3. 逐点相乘. 把 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 逐点相乘，得到点值向量 $\boldsymbol{C}$.
4. 插值. 调用输入规模为 n 的 IFFT 算法，计算 $\boldsymbol{C}$ 对应的系数向量 $\boldsymbol{c}$.
5. 返回向量 $\boldsymbol{c}$.

算法 2.　NTT-PMA($\boldsymbol{a},\boldsymbol{b},N$).

输入：$\boldsymbol{a},\boldsymbol{b},N$. 其中 $\boldsymbol{a},\boldsymbol{b}$ 是系数向量，N 是次数界

输出：$c(x)$ 的系数向量 $\boldsymbol{c}$，满足 $c(x)=a(x)\cdot b(x)$

1. 加倍次数界. 在向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的高位添加值为 0 元素，直至向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的长度为 $n=2^{\lceil \log N \rceil+1}$.
2. 求值. 调用输入规模为 n 的 NTT 算法，计算 $\boldsymbol{a}$ 对应的点值向量 $\boldsymbol{A}$，和 $\boldsymbol{b}$ 对应的点值向量 $\boldsymbol{B}$.
3. 逐点相乘. 把 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 逐点模乘，得到点值向量 $\boldsymbol{C}$.
4. 插值. 调用输入规模为 n 的 INTT 算法，计算 $\boldsymbol{C}$ 对应的系数向量 $\boldsymbol{c}$.
5. 返回向量 $\boldsymbol{c}$.

正向 NTT 是系数向量 $\boldsymbol{a}=(a_0,\cdots,a_{n-1})$ 到点值向量 $\boldsymbol{A}=(A_0,\cdots,A_{n-1})$ 的映射，记 $\boldsymbol{A}=\mathrm{NTT}_\alpha(\boldsymbol{a})$，其中

$$A_i=\sum_{j=0}^{n-1}a_j\alpha^{ij} \bmod M,\ i=0,1,\cdots,n-1,$$

记 $\boldsymbol{a}=\mathrm{INTT}_\alpha(\boldsymbol{A})$ 为 NTT 的逆，其中

$$a_i=n^{-1}\sum_{j=0}^{n-1}A_j\alpha^{-ij} \bmod M,\ i=0,1,\cdots,n-1.$$

定理 1.　折叠卷积[25]. 设 α 是 $\mathbb{Z}_M$ 中的 n 次单位根，$\varphi^2=\alpha$；设向量 $\boldsymbol{a}=(a_0,a_1,\cdots,a_{n-1})\in\mathbb{Z}_M^n$，向量 $\boldsymbol{b}=(b_0,b_1,\cdots,b_{n-1})\in\mathbb{Z}_M^n$；$\circ$ 表示逐点相乘.

(1) $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的正折叠卷积为

$$\mathrm{INTT}_\alpha(\mathrm{NTT}_\alpha(\boldsymbol{a})\circ\mathrm{NTT}_\alpha(\boldsymbol{b})).$$

(2) 设 $\boldsymbol{c}=(c_0,\cdots,c_{n-1})$ 是 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的负折叠卷积，定义

$$\bar{\boldsymbol{a}}=(a_0,\varphi\cdot a_1,\varphi^2\cdot a_2,\cdots,\varphi^{n-1}\cdot a_{n-1}),$$

$$\bar{\boldsymbol{b}}=(b_0,\varphi\cdot b_1,\varphi^2\cdot b_2,\cdots,\varphi^{n-1}\cdot b_{n-1}),$$

$$\bar{\boldsymbol{c}}=(c_0,\varphi\cdot c_1,\varphi^2\cdot c_2,\cdots,\varphi^{n-1}\cdot c_{n-1}),$$

则 $\bar{\boldsymbol{c}}=\mathrm{INTT}_\alpha(\mathrm{NTT}_\alpha(\bar{\boldsymbol{a}})\circ\mathrm{NTT}_\alpha(\bar{\boldsymbol{b}}))$.

多项式 a 和 b 在 $\mathbb{Z}_M[x]/(x^n-1)$ 上的乘积等于 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的正折叠卷积，在 $\mathbb{Z}_M[x]/(x^n+1)$ 上的乘积等于 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的负折叠卷积. 格密码方案通常取特殊环 $\mathbb{Z}_q[x]/(x^N+1)$ 且 N 为 2 的方幂，利用 NTT 负折叠卷积可以避免加倍次数界，提升多项式乘法效率. 基于 NTT 负折叠卷积的多项式乘法（NTT-Negative Wrapped Convolution，NTT-NWC）见算法 3，其中 NTT-BAR 及其逆算法 INTT-BAR 见附录 A.

算法 3.　NTT-NWC($\boldsymbol{a},\boldsymbol{b},N$).

输入：$\boldsymbol{a},\boldsymbol{b},N$. 其中 $\boldsymbol{a},\boldsymbol{b}$ 是系数向量，次数界 $N=2^k$

输出：$c(x)$ 的系数向量 $\boldsymbol{c}$，满足 $c(x)=a(x)\cdot b(x)$

1. 求值. 调用输入规模为 N 的 NTT-BAR 算法，得到 $\bar{\boldsymbol{A}}=\mathrm{NTT}_\alpha(\bar{\boldsymbol{a}})$ 和 $\bar{\boldsymbol{B}}=\mathrm{NTT}_\alpha(\bar{\boldsymbol{b}})$.
2. 逐点相乘. 计算 $\bar{\boldsymbol{C}}=\bar{\boldsymbol{A}}\circ\bar{\boldsymbol{B}}$
3. 插值. 调用输入规模为 N 的 INTT-BAR 算法，得到 $\boldsymbol{c}=(1,\varphi_1^{-1},\cdots,\varphi^{-n+1})\circ\mathrm{INTT}_\alpha(\bar{\boldsymbol{C}})$.
4. 返回向量 $\boldsymbol{c}$.

2.4　模整数算法

Montgomery 算法和 Barrett 算法是通用模整数算法，二者的本质都是在预计算的基础上，利用加减、乘法和移位代替除法. Montgomery 算法分为预计算和约化两部分，具体描述见附录 B 中算法 B1，预计算时需选择一个正整数 r，为提高计算效率 r 通常取 2 的方幂. 由于 Montgomery 算法实际输出

的是 $r^{-1}ab(\mathrm{mod}M)$，因此需进行预计算和后处理来去除 r^{-1} 的影响. Barrett 算法分为预计算和约化两部分，具体描述见附录 B 中算法 B2. 其基本思想是，将取模运算中的除法求商运算 $\lfloor x/M \rfloor$ 转化为用预先计算好的模的倒数 $\lfloor 2^{2k}/M \rfloor$ 与输入相乘的求商运算，k 的取值不唯一，通常取 CPU 字长的整数倍.

除通用模整数算法外，当模数取特殊值时，可以采用特殊模整数方法：模数为 2^t 时，可以用按位与操作进行模运算. 模数为 2^t+1 或 2^t-1 时，可以用加减、按位与、移位进行模运算，见附录 B.

3 相关工作

现有格密码方案的多项式环大多取特殊环 $\mathbb{Z}_q[x]/(x^N+1)$，对于这类特殊环，多项式乘法的耗时远高于模约化和舍入计算. 根据现有的 RLWR 和 MLWR 方案实现及效率优化工作，多项式乘法的运行时间通常占 RLWR 或 MLWR 总运行时间的 95%左右，因此多项式乘法的效率决定了 RLWR 和 MLWR 的效率；此外，格密码领域多项式乘法实现方案较多，故 3.1 节单独讨论多项式乘法实现. RLWR 和 MLWR 实现方案见 3.2 节.

3.1 格密码领域多项式乘法实现方案

整数域上通用多项式乘法算法及其理论时间复杂度见表 1. 在一定条件下，基于 FFT 和 NTT 的多项式乘法算法的时间复杂度可以达到目前的理论下界，但由于 FFT 算法涉及到复数运算，而 NTT 只需执行模整数运算，二者的实际效率可能存在差异，通常情况下，通用 CPU 平台模整数运算的效率高于复数运算；此外，计算特殊环 $\mathbb{Z}_q[x]/(x^N+1)$ 上的多项式乘法时，可利用负折叠卷积定理将 NTT 乘法算法的输入规模减半，此时 NTT 负折叠卷积的效率高于 FFT 乘法算法. 故对于实用的格密码方案，与其他通用多项式乘法算法相比，基于 NTT 的多项式乘法算法的效率相对较高.

表 1 通用多项式乘法算法及其理论时间复杂度

多项式乘法算法	理论时间复杂度
笔乘/Comba[26]	$O(n^2)$
Karatsuba/Toom-Cook(2-way)	$O(n^{1.58})$
Toom-Cook(k-way)	$O(n^{\log(2k-1)/\log k})$
SSA[27]	$O(n\cdot\log n\cdot\log\log n)$
Fürer[28]	$O(n\cdot\log n\cdot 2^{\mathcal{O}(\log^* n)})$
FFT/NTT	$\Omega(n\cdot\log n)$
HHA[29]	$O(n\cdot\log n\cdot 4^{\log^* n})$

注：n 表示输入多项式的次数界；log 底为 2，$\log^* n$ 表示迭代对数.

Kyber、Dilithium 和 qTESLA 等方案采用 NTT 乘法算法，且都借助负折叠卷积定理将 NTT 输入规模减半，并通过预计算旋转因子等变量进一步提升效率. 虽然上述三个方案的参数不同，但模数 q 都是 NTT 模数，环均为 $\mathbb{Z}_q[x]/(x^N+1)$. 由于旋转因子由 NTT 模数和输入规模决定，故上述 NTT 实现无法应用于参数 N 和 q 取其他值的密码方案.

Saber方案使用 Karatsuba 和 Toom-Cook(4-way) 乘法算法. 为提升模约化、舍入计算和随机数采样的效率，Saber 方案选取 2 的方幂作为模数 q. Saber 团队指出，使用这种模数的主要缺点是无法采用 NTT 加速多项式乘法，考虑到 Saber 方案多项式次数界较小($N=256$)，可以通过组合 Karatsuba 与 Toom-Cook 算法提升乘法效率. 但二者的渐近时间复杂度较高，其效率随次数界的增加显著降低，因此不适用于参数 N 较大($N\geq 512$)的方案实现.

事实上，若参数 N，q，S 较小，使得存在 NTT 模数 M 满足 $qSN<2M$，即使 q 不满足 NTT 模数的一般性要求，也可以采用 NTT 计算多项式乘法. Saber 优化实现[21] 采用了 NTT 乘法算法，优化后密钥封装的效率在 Cortex-M4 平台提高了 22%左右. 但该工作仅适用于 Saber 实现，且未给出 NTT 的使用条件与格密码参数和 CPU 字长的量化关系.

LIMA 等方案采用 FFT 乘法算法. 基于 FFT 的乘法算法可以计算任意两个多项式的乘积，但 FFT 涉及浮点数运算，故 FFT 乘法算法的效率通常低于 NTT 乘法算法. 此外，浮点数运算可能存在误差.

Round5 和 Lizard 等方案采用稀疏乘法算法. s 的系数较稀疏，即 $\mathrm{P}[s_i\neq 0]$ 较小时，可考虑稀疏乘法算法，本质上是利用滑动窗口的思想，采用加减运算计算多项式乘法，见算法 4，该算法只适用于 $S<4$ 的方案实现，即 s 系数取自集合 $\{-1,0,1\}$. 稀疏乘法不是常数时间算法，其运行时间与 $\mathrm{P}[s_i\neq 0]$ 成正比，且易受电磁分析攻击等侧信道攻击[30].

算法 4. SPMA($\boldsymbol{a}$,$\boldsymbol{b}$,N).

输入：向量 $\boldsymbol{a}$,$\boldsymbol{b}$，正整数 N. $\boldsymbol{b}$ 中元素取自集合 $\{-1,0,1\}$

输出：$c(x)$ 的系数向量 $\boldsymbol{c}$，满足 $c(x)=a(x)\cdot b(x)$

```
1. c[2N]={0}
2. FOR i=0 TO N-1 DO
3.     IF b[i]==1 THEN
4.         FOR j=0 TO N-1 DO
5.             c[i+j]=c[i+j]+a[j]
6.     ELSE IF b[i]==-1 THEN
7.         FOR j=0 TO N-1 DO
8.             c[i+j]=c[i+j]-a[j]
9. RETURN c
```

综上所述，NTT 乘法算法的理论复杂度最低，实际效率相对较高，但格密码领域现有的 NTT 实现大多针对特定密码方案，不能处理任意参数；此外，格密码领域对 NTT 实际使用条件的研究较少，NTT 的限制条件与格密码参数 N,f,q,S 和 CPU 字长 B 之间的关系尚待研究.

3.2　RLWR 和 MLWR 实现方案

目前，RLWR 和 MLWR 主要应用于构造伪随机函数、陷门函数、公钥加密以及全同态加密. 然而，除公钥加密外，其他方案大多是理论构造，未提供方案实现. 现有基于 RLWR 或 MLWR 的公钥加密方案主要包括 Lizard、Round5 和 Saber，三者都是 NIST 后量子密码标准算法征集活动中的提案.

Lizard 提案中的 RLizard 方案是基于 RLWR 的密码方案. RLizard 方案参数 $S<4$，多项式乘法采用稀疏乘法算法；参数 f 为 x^N-1，模多项式采用减法；参数 p 和 q 均为 2 的方幂，模整数采用按位与操作，舍入计算采用先加法再移位. 故 RLizard 实现只适用于 $f(x)=x^N+1$，参数 p 和 q 均为 2 的方幂，且 $S<4$ 的 RLWR.

Round5 提案中的 R5ND 方案是基于 RLWR 的密码方案. 该方案参数 f 为 $x^N+x^{N-1}+\cdots+1$，由于 $(x-1)\cdot f(x)=x^{N+1}-1$，为提升实现效率，R5ND 将 $\mathbb{Z}[x]/(f)$ 上的运算映射至环 $\mathbb{Z}[x]/(x^{N+1}-1)$ 上进行. $\mathbb{Z}[x]/(f)$ 上多项式乘法 $a\cdot s$ 大致分为三步：

① $a^{(L)}=(x-1)\cdot a$，通过乘 $x-1$，将 $\mathbb{Z}[x]/(f)$ 上的 a 提升至 $\mathbb{Z}[x]/(x^{N+1}-1)$ 上，可采用加减法实现；

② $c^{(L)}=(a^{(L)}\cdot s \bmod (x^{N+1}-1))$，R5ND 方案参数 $S<4$，多项式乘法采用稀疏乘法算法，模多项式采用加减法实现；

③ $c=c^{(L)}/(x-1)$，通过除 $x-1$，将 $c^{(L)}$ 还原至 $\mathbb{Z}[x]/(f)$ 上，此步骤采用加减法实现.

R5ND 方案参数 p 和 q 均为 2 的方幂，模整数采用按位与操作，舍入计算采用先加法再移位. 故 R5ND 实现只适用于 $f(x)=x^N+x^{N-1}+\cdots+1$，$p$ 和 q 均为 2 的方幂，且 $S<4$ 的 RLWR.

Saber 方案是基于 MLWR 的密码方案. Saber 的多项式乘法采用 Karatsuba 和 Toom-Cook 算法；Saber 方案参数 f 为 x^N+1，模多项式采用减法；参数 p 和 q 均为 2 的方幂，模整数采用按位与操作，舍入计算采用先加法再移位. 故 Saber 实现只适用于 $f(x)=x^N+1$，p 和 q 均为 2 的方幂的 MLWR.

综上所述，现有 RLWR 和 MLWR 实现大多无法处理任意参数，且大多未采用效率相对较高的 NTT 类多项式乘法算法；同时，现有 RLWR 和 MLWR 参数 p 和 q 通常直接取 2 的方幂，缺乏系统的 RLWR 和 MLWR 快速实现参数选取方法. 考虑到 Saber 方案是 NIST 决赛入围算法，进入了第三轮评估，而 Lizard 方案未入选第二轮评估，Round5 方案未入选第三轮评估，故本文重点关注 Saber 方案.

4　通用 RLWR 和 MLWR 实现算法

本节提出通用的 RLWR 实现算法和 MLWR 实现算法. RLWR/MLWR 实现算法根据方案参数和运行平台的 CPU 字长，灵活选取高效的多项式乘法、模约化和舍入计算实现方案. 具体地，本文提出的通用 RLWR/MLWR 实现算法分为初始化模块和计算模块两部分，初始化模块用于选择三项基础操作的实现方案，计算模块用于计算 RLWR/MLWR 分布中的 $b/\boldsymbol{b}$. 其中，初始化模块调用了多项式乘法初始化算法、模约化初始化算法以及舍入计算初始化算法. 本节默认实验环境为 64 位 Intel(R) Core(TM) i7-6700 CPU@3.40GHz，ubuntu16.04，GCC 编译器，未使用编译器优化指令.

4.1　多项式乘法初始化算法

本小节提出多项式乘法初始化算法，输入方案参数和 CPU 字长，该算法根据 FFT 乘法算法、NTT 乘法算法、NTT 负折叠卷积和稀疏乘法算法的使用条件，确定可以使用的多项式乘法算法，然后输出其中效率相对较高的多项式乘法实现方案名称. 本小节首先讨论 NTT 乘法算法和 NTT 负折叠卷积的使用条件，其次测试各多项式乘法算法的效率，最后给出多项式乘法初始化算法.

理论上，NTT 乘法算法可以计算任意两个多项式的乘积，模数为 M 的 NTT 乘法算法的输出是 $\mathbb{Z}_M=\left\{-\left\lfloor\frac{M-1}{2}\right\rfloor,\cdots,0,\cdots,\left\lfloor\frac{M}{2}\right\rfloor\right\}$ 上的乘积多项式，若 M 足够大，则可认为输出即是整数域上的结果. NTT 算法主要包括整数的加减法、乘法和模运算，其中间值的值域为 $\left[-2\times\left\lfloor\frac{M-1}{2}\right\rfloor\times\left\lfloor\frac{M}{2}\right\rfloor,2\times\left\lfloor\frac{M}{2}\right\rfloor^2\right]$. 考虑到计算机字长是有限的，CPU 一次操作中能处理的整数存在上界和下界，因此实际中 NTT 模数存在上界. B 位通用 CPU 平台能直接处理的有符号整数最大区间是 $[-2^{B-1},2^{B-1}-1]$，因此 NTT 模数 M 需满足

$$\left[-2\times\left\lfloor\frac{M-1}{2}\right\rfloor\times\left\lfloor\frac{M}{2}\right\rfloor,2\times\left\lfloor\frac{M}{2}\right\rfloor^2\right]\subseteq[-2^{B-1},2^{B-1}-1],$$

即 $M<2^{B/2}$. 目前大部分商用计算机可支持 64 位处理

器,本文重点关注 64 位通用 CPU 平台,该平台上 NTT 模数需满足 $M<2^{32}$.

记 $p_1,\cdots,p_t$ 是 M 的全部素因子,N 为输入多项式的次数界,可在 B 位 CPU 平台高效实现的 NTT,其输入规模 n,模数 M 和单位根 α 需满足[15]:

① n 是 2 的方幂. 通常取 $n=2^{\lceil\log N\rceil+1}$.

② $M\in \mathrm{M}(n,B)$,其中,集合 $\mathrm{M}(n,B)$ 定义为

$$\mathrm{M}(n,B)=\{M:0<M<2^{B/2},\gcd(n,M)=1,\ n\mid\gcd(p_1-1,\cdots,p_t-1)\}.$$

③ α 满足 $\gcd(\alpha,M)=1$ 且 n 是使 $\alpha^n\equiv1(\mathrm{mod}\,M)$ 成立的最小正整数.

记 $\mathrm{M}_{\max}(n,B)$ 为集合 $\mathrm{M}(n,B)$ 中的最大值,其含义是可在 B 位 CPU 平台实现的规模为 n 的 NTT 模数最大值. $\mathrm{M}_{\max}(n,64)$ 的部分取值见表 2. $\mathrm{M}_{\max}(n,B)$ 计算方法见算法 5.

表 2 64 位 CPU 平台 NTT 模数最大值

N	n	$\mathrm{M}_{\max}(n,64)$
$2^7<N\le2^8$	2^9	$4\,294\,962\,689(=2^9\times17\times493\,447+1)$
$2^8<N\le2^9$	2^{10}	$4\,294\,962\,689(=2^9\times17\times493\,447+1)$
$2^9<N\le2^{10}$	2^{11}	$4\,294\,957\,057(=2^{11}\times3\times13\times53\,773+1)$
$2^{10}<N\le2^{11}$	2^{12}	$4\,294\,957\,057(=2^{11}\times3\times13\times53\,773+1)$
$2^{11}<N\le2^{12}$	2^{13}	$4\,294\,955\,009(=2^{11}\times1\,048\,573+1)$

注:N 是 NTT 乘法算法输入多项式的次数界,可以是任意正整数;n 是对应的 NTT 乘法算法输入规模. 表中 $\mathrm{M}_{\max}(n,64)$ 的值均为素数.

算法 5. $\mathrm{M}_{\max}(n,B)$.

输入:正整数 n,B. n,B 均为 2 的方幂

输出:正整数 M. $M=\mathrm{M}_{\max}(n,B)$

```
1.  M=2^{B/2}-1
2.  WHILE M>0 DO
3.      s=0
4.      计算 M 的所有素因子 m_1,m_2,…,m_t
5.      FOR temp=m_1,m_2,…,m_t DO
6.          IF (temp-1)%n!=0 THEN
7.              s=1
8.              BREAK
9.      IF s==0 THEN
10.         BREAK
11.     ELSE M=M-2
12. END WHILE
13. RETURN M
```

NTT 乘法算法的使用条件与 RLWR/MLWR 参数 N,q,S 有关,对于以下两类参数,可在 B 位通用 CPU 平台采用基于 NTT 的多项式乘法算法:

(1) $q\in\mathrm{M}(2^{\lceil\log N\rceil+1},B)$,即参数 q 满足输入规模为 $2^{\lceil\log N\rceil+1}$ 的 NTT 模数的限制条件. 选择 q 作为 NTT 模数,采用 NTT 乘法算法可直接得到 $\mathbb{Z}_q$ 上的乘积多项式,此时可省略后续模整数操作;

(2) $qSN<2\mathrm{M}_{\max}(2^{\lceil\log N\rceil+1},B)$,即参数 N,q,S 足够小,使得乘积多项式的系数都在 $\mathbb{Z}_M$ 内.

NTT 负折叠卷积的输入规模 n,模数 M 和单位根 α 的限制条件与 NTT 中三个参数的限制条件基本相同,不同之处在于条件①中取 $n=2^{\log N}$. NTT 负折叠卷积只能用于计算特殊环 $\mathbb{Z}_M[x]/(x^N+1)$ 上的多项式乘法,其使用条件与 RLWR/MLWR 参数 N,f,q,S 有关. 令 $k=\lceil\log N\rceil$,对于以下两类参数,可在 B 位通用 CPU 平台采用基于 NTT 负折叠卷积的多项式乘法算法:

(1) $N=2^k$, $f(x)=x^N+1$, $q\in\mathrm{M}(N,B)$;此时可省略后续模多项式和模整数操作;

(2) $N=2^k$, $f(x)=x^N+1$, $qSN<2\mathrm{M}_{\max}(N,B)$;此时可省略后续模多项式操作.

NTT 和 NTT 负折叠卷积的使用条件与参数 N,f,q,S 和 CPU 字长 B 之间的关系见表 3. NTT 输入规模 n 和 NTT 模数 M 会影响 RLWR/MLWR 实现效率:输入规模 n 决定 NTT 和 NTT 负折叠卷积的效率,模数 M 影响后续模约化的效率.

表 3 *B* 位 CPU 平台 NTT 类乘法算法使用条件

NTT 类乘法算法 名称	模数	n	使用条件	后续模约化
NTT 负折叠卷积	q	2^k	① $N=2^k$ ② $f(x)=x^N+1$ ③ $q\in\mathrm{M}(N,B)$	无
	M	2^k	① $N=2^k$ ② $f(x)=x^N+1$ ③ $qSN<2\mathrm{M}_{\max}(N,B)$	模 q
NTT 乘法算法	q	2^{k+1}	① $q\in\mathrm{M}(n,B)$	模 $f(x)$
	M	2^{k+1}	① $qSN<2\mathrm{M}_{\max}(n,B)$	模 $f(x)$,模 q

注:N 是输入多项式的次数界,$k=\lceil\log N\rceil$,n 为算法的输入规模,M 表示任意满足条件的 NTT 模数. RLWR/MLWR 参数定义见 2.2 节.

本文测试了不采用并行运算的情况下,常用多项式乘法快速实现算法的效率,见表 4,其中,NTT 负折叠卷积、NTT 多项式乘法算法、稀疏乘法以及 FFT 多项式乘法算法的效率较高,故多项式乘法实现方案从上述四个算法中选取.

表 4 常用多项式乘法快速实现算法运行时间

多项式乘法实现算法	多项式乘法运行时间/μs		
	$N=256$	$N=512$	$N=1024$
笔乘/Comba	196	731	2965
Karatsuba/Toom-Cook(2-way)	409	1221	3682
FFT 乘法算法	88	194	427
稀疏乘法算法($\mathrm{P}[s_i\neq0]=0.25$)	42	172	712
NTT 乘法算法	43	97	198
NTT 负折叠卷积	21	44	93

注:N 是输入多项式的次数界,FFT、NTT、NTT 负折叠卷积的实现均预计算了旋转因子等变量. NTT、NTT 负折叠卷积的模数均取 $119\times2^{23}+1$.

本文对比了 64 位通用 CPU 平台，输入多项式的次数界 $N=256$，参数 $\mathrm{P}[s_i\neq 0]$取不同值时，上述四个效率较高的多项式乘法算法的效率，见图 2. NTT 负折叠卷积、NTT 多项式乘法算法以及 FFT 多项式乘法算法的效率由参数 N 决定，N 同时满足三者的使用条件时，上述三个算法的效率依次递减；稀疏乘法算法的效率由输入多项式的次数界和多项式 s 零系数的数量共同决定，具体地，其运行时间与 N^2 和 $\mathrm{P}[s_i\neq 0]$成正比. 令 $k=\lceil\log N\rceil$，根据本文的实验结果，$\mathrm{P}[s_i\neq 0]<2^{k+7}/N^2$ 时，稀疏乘法算法的效率高于 FFT 乘法算法；$\mathrm{P}[s_i\neq 0]<2^{k+6}/N^2$ 时，稀疏乘法算法的效率高于 NTT 乘法算法；$\mathrm{P}[s_i\neq 0]<1/2^{k-5}$ 时，稀疏乘法算法的效率高于 NTT 负折叠卷积.

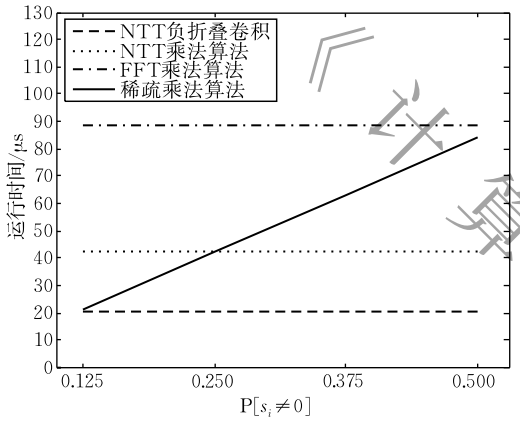

图 2　多项式乘法算法运行时间($N=256$)

本文提出多项式乘法初始化算法(PMA_Setup)，见算法 6，该算法适用于 64 位通用 CPU 平台. 该算法首先根据方案参数 $N,f,q,S,\mathrm{P}[s_i\neq 0]$和 CPU 字长 B，确定可以使用的多项式乘法算法，然后选择其中效率相对较高的算法作为乘法实现方案. 不同运行平台乘法算法的实际效率可能存在差异，在其他 CPU 平台实现并对比上述四个乘法算法的效率后，算法 6 即可推广至任意 CPU 平台. 注意，稀疏乘法算法易受侧信道攻击，若多项式 s 包含秘密信息，则应尽量避免采用稀疏乘法算法.

算法 6.　PMA_Setup(N,f,q,S,B,P).

输入：正整数 N,q,S,B，浮点数 P，整数数组 $\boldsymbol{f}$. 其中 P 即参数 $\mathrm{P}[s_i\neq 0]$，$\boldsymbol{f}$ 即环元素 f 的系数向量

输出：字符串 *PMA*. 其含义为 RLWR/MLWR 计算模块采用的多项式乘法实现算法名称

1. $k=\lceil\log N\rceil$
2. 计算 q 的所有素因子 $q_1,\cdots,q_t$
3. $Oq=\gcd(q_1-1,\cdots,q_t-1)$ //$q_1-1,\cdots,q_t-1$ 的最大公因子
4. IF ($f[N]==1$) AND ($f[0]==1$) THEN
5. 　FOR $i=1$ TO $N-1$ DO
6. 　　IF $f[i]!=0$ THEN
7. 　　$temp=1$
8. 　　BREAK
9. 　　ELSE $temp=0$ //$temp=0$ 表示$f(x)=x^N+1$
10. ELSE $temp=1$
11. IF ($N==2^k$) AND ($temp==0$) THEN
12. 　IF ($S<4$) AND ($P<=1/2^{k-5}$) THEN
13. 　　RETURN “SPMA”
14. 　ELSE IF ($Oq\%N==0$ AND ($q<2^{B/2}$) THEN
15. 　　RETURN “NTT-NWC_q, N” //表示采用 NTT-NWC 算法，NTT 模数取 q，输入规模为 N
16. 　ELSE IF $q*S*N<2*\mathrm{M}_{\max}(2^k,B)$ THEN
17. 　　RETURN “NTT-NWC_M, N” //表示采用 NTT-NWC 算法，NTT 模数不能取 q，需自行选择一个合适的 NTT 模数 M，输入规模为 N
18. 　ELSE IF ($S<4$) AND ($P<=2^{k+7}/N^2$) THEN
19. 　　RETURN “SPMA”
20. 　ELSE RETURN “FFT-PMA”
21. ELSE IF ($S<4$) AND ($P<=2^{k+6}/N^2$) THEN
22. 　RETURN “SPMA”
23. ELSE IF ($Oq\%2^{k+1}==0$ AND ($q<2^{B/2}$) THEN
24. 　RETURN “NTT-PMA_q, 2^{k+1}” //表示采用 NTT-PMA 算法，NTT 模数取 q，输入规模为 2^{k+1}
25. ELSE IF $q*S*N<2*\mathrm{M}_{\max}(2^{k+1},B)$ THEN
26. 　RETURN “NTT-PMA_M, 2^{k+1}” //表示采用 NTT-PMA 算法，NTT 模数不能取 q，需自行选择一个合适的 NTT 模数 M，输入规模为 2^{k+1}
27. ELSE IF ($S<4$) AND ($P<=2^{k+7}/N^2$) THEN
28. 　RETURN “SPMA”
29. ELSE RETURN “FFT-PMA”

4.2　模约化初始化算法

本小节提出模约化初始化算法，输入参数 f,q 和已选取的多项式乘法实现方案名称，该算法输出效率相对较高的模多项式和模整数实现方案名称. 本小节首先讨论现有模多项式实现方法的使用条件及效率，其次讨论现有模整数实现方法的使用条件及效率，最后给出模约化初始化算法.

带余除法是通用的模多项式实现方法，但该方法需执行大量除法运算，效率较低；f 为 N 次首一多项式时，模 f 可用 N^2 次减法和乘法实现，见附录 C 中算法 C1(Subtraction and Multiplication-Modular Polynomial Algorithm, SM-MPA)；格密码领域 f 大多取 x^N+1，此时模 f 可用 N 次减法实现，见附录 C 中算法 C2(Subtraction-MPA, Sub-MPA). 上述模多项式算法的效率见表 5，可以看出，带余除法的效率显著低于其他算法. 格密码领域 f 取非首一多项式的密码方案较少，故本文假设 f 为首一多项

式，模多项式实现方案采用 SM-MPA 或 Sub-MPA，根据二者的实现效率，若 $f(x)=x^N+1$，则模多项式实现方案采用 Sub-MPA，否则采用 SM-MPA.

表 5 模多项式运行时间

$f(x)$	模$f(x)$实现算法	运行时间/μs		
		$N=256$	$N=512$	$N=1024$
$\sum_{i=0}^{N} f_i x^i$	带余除法	1.3×10^4	9.9×10^4	5.9×10^5
$x^N+\sum_{i=0}^{N-1} f_i x^i$	C1. SM-MPA	2.1×10^2	7.9×10^2	3.2×10^3
x^N+1	C2. Sub-MPA	0.7	1.4	2.9

常用模整数算法实现效率见表 6，其中模数 q 作为变量参与运算，编译器未优化模运算. 可以看出，模数 q 作为变量参与运算时，效率较高的模整数算法是 Barrett 算法和按位与，无优化模整数(C 语言中的%运算符)的运行时间显著高于其他算法.

表 6 模整数(变量)运行时间

模整数算法	模变量运行时间/μs		
	$N=256$	$N=512$	$N=1024$
无优化模运算(%运算符)	1.91	3.79	7.69
Montgomery	0.62	1.23	2.57
Barrett	0.61	1.20	2.53
按位与	0.46	0.91	1.99
算法 B3/算法 B4	0.68	1.39	2.79

然而，实际中格密码方案的模数通常是预先选定的常数，模数作为常量参与运算. GCC 编译器会优化常量的模运算：模数 q 为 2 的方幂时，采用按位与或取低位比特进行模运算；模数 q 不是 2 的方幂时，采用 Barrett 算法进行模运算. 因此，在不直接使用汇编代码的情况下，采用 Barrett 等优化算法反而会产生额外的存取负担. 模常量实现方法的效率对比见表 7，可以看出按位与的效率相对较高，其次是无优化模运算. 因此，模整数实现方案采用按位与或无优化模运算，根据二者的实现效率，若 q 为 2 的方幂，则模整数实现方案采用按位与，否则采用无优化模运算.

表 7 模整数(常量)运行时间

模整数算法	$N=1024$ 模常量 q 运行时间/μs		
	$q=33215$	$q=2^{16}$	$q=2^{16}\pm1$
无优化模运算(%运算符)	2.47	2.20	2.04
Montgomery	2.59	2.58	2.64
Barrett	2.48	2.46	2.59
按位与	/	2.17	/
算法 B3/算法 B4	/	/	2.43

注：$2^{16}+1$ 是 NTT 模数.

模约化初始化算法(Modular Reduction Algorithm_Setup，MRA_Setup)见算法 7，该算法根据参数 f，q 和已选取的多项式乘法实现方案 PMA，选择适用且高效的模多项式和模整数实现方案. 具体地，当采用某些特殊多项式乘法实现方案时，可省略模多项式或模整数操作；否则，模多项式采用 SM-MPA 或 Sub-MPA，模整数采用无优化的模运算(Mod-Modular Integer Algorithm，Mod-MIA)或按位与(And-MIA).

算法 7. MRA_Setup(q, $\boldsymbol{f}$, PMA).

输入：正整数 q，整数数组 $\boldsymbol{f}$，字符串 PMA

输出：字符串数组 ***MRA***. 其含义为 RLWR/MLWR 计算模块采用的模约化实现算法，$MRA[0]$对应模多项式实现算法名称，$MRA[1]$对应模整数实现算法名称

```
1.  n=f.length
2.  IF (PMA=="NTT-NWC_q,N") OR (PMA=="NTT-NWC_M,N") THEN
3.    MRA[0]="N-MPA"//N-MPA表示无需模多项式
4.  ELSE
5.    IF (f[n]==1) AND (f[0]==1) THEN
6.      FOR i=1 TO N-1 DO
7.        IF f[i]!=0 THEN
8.          temp=1
9.          BREAK
10.       ELSE temp=0 //temp=0表示f(x)=x^N+1
11.     ELSE temp=1
12.   IF temp==0 THEN
13.     MRA[0]="Sub-MPA"
14.   ELSE MRA[0]="SM-MPA"
15. IF (PMA=="NTT-NWC_q,N") OR (PMA=="NTT-PMA_q,2^(k+1)") THEN
16.   MRA[1]="N-MIA" //N-MIA 表示无需模整数
17. ELSE
18.   t=⌈logq⌉
19.   IF q==2^t THEN
20.     MRA[1]="And-MIA"
21.   ELSE MRA[1]="Mod-MIA"
22. RETURN MRA
```

4.3 新舍入算法及舍入计算初始化算法

舍入计算是格密码领域的基础运算之一，其主要的计算开销在于除法. 本小节提出舍入计算初始化算法，输入参数 p 和 q，该算法输出适用且高效的舍入算法名称. 本小节首先提出一个新舍入算法，其次测试现有舍入实现方法以及新舍入算法的实现效率，最后给出舍入计算初始化算法.

先加法再移位是目前最高效的舍入实现方法，但只适用于比值 q/p 为 2 的方幂的方案；而部分格方案模数 q 取奇素数，故无法采用该方法加速舍入计算. 但本文发现，模数取形式为 $q=c\times2^t+1$ 的

NTT 模数时，选取合适的 p 可满足 $(q-1)/p=2^t$，虽然先加法再移位无法得到准确的舍入结果，但仅有少量输入的舍入值存在误差，该误差可通过一次减法运算修正. 考虑到 p 的值影响 LWR 安全性，若 $p=(q-1)/2^t$，那么只要取足够大的 t 使 $2^t\geq 2mE$ 成立，即可完成多项式级别模数下 LWE 到 LWR 的归约，其中 m 为采样个数，E 为 LWE 误差边界.

基于上述思想，本文设计了一种新舍入实现算法，其核心是在预计算的基础上，利用移位和加减法代替除法运算：当 q 是奇素数，且 q 是输入规模为 t 的 NTT(或 NTT 负折叠卷积)模数时，q 可以表示为 $c\cdot 2^t+1$，其中正整数 $t\in\left[\log N+1,\frac{B}{2}-\log c\right)$ (或 $t\in\left[\log N,\frac{B}{2}-\log c\right)$)，$c$ 是正整数. 对于取 $p=c$ 的方案，可以采用移位和加减法实现舍入计算. 具体地，$\mathbb{Z}_q$ 中某些元素先做加法再右移 t 位后的值不等于其舍入值，本文称这些元素为误差输入点(简称误差点)，$\mathbb{Z}_q$ 中共有 p 个误差点，这些误差点的集合为 $\{2^{t-1}(2i+1)\}_{0\leq i\leq p-1,i\in\mathbb{Z}}$，误差为 1，其余输入点的误差视为 0，可提前计算并存储每个点的误差值. 实际舍入计算时，对某个输入点，做完加法和移位后，减去该点误差即可得到精确舍入值.

新舍入算法(New-Rounding Algorithm，New-RA)见算法 8，该算法的存储开销至少为 q 比特. 为使算法描述简洁，本小节假舍入算法的输入 x 取自最小非负剩余系 $\{0,\cdots,q-1\}$.

算法 8.　New-RA(x,p,q,t).

输入：自然数 x，正整数 p,q,t；$q=p\times 2^t+1$，$x<q$

输出：舍入值 u；$u=\lfloor (p/q)\cdot x\rceil \bmod p$

预计算

1. Pre$[q]=\{0\}$
2. FOR $i=0$ TO $p-1$ DO
3. 　Pre$[(2i+1)\ll(t-1)]=1$

舍入计算

1. $u=\lfloor (x+2^{t-1})\gg t\rfloor-Pre[x]$
2. RETURN u

正确性分析：记 $y_x=\frac{p}{q}\cdot x+\frac{1}{2}$，$y_x^*=\frac{1}{2^t}\cdot x+\frac{1}{2}$，有 $y_x<y_x^*$；记整数 u_x 为 x 的精确舍入值，整数 u_x^* 为 x 先加法再移位后可能存在误差的舍入值，有 $u_x=\lfloor y_x\rfloor\leq y_x$，$u_x^*=\lfloor y_x^*\rfloor\leq y_x^*$；记误差点的集合为 $\mathrm{W}=\{2^{t-1}(2i+1)\}_{0\leq i\leq p-1,i\in\mathbb{Z}}$. 对于舍入算法的输入 $x\in\mathbb{Z}_q=\{0,\cdots,q-1\}$，下面我们分别证明①$x\in\mathrm{W}$ 时，$u_x=u_x^*-1$；②$x\notin\mathrm{W}$ 时，$u_x=u_x^*$.

①$x\in\mathrm{W}$. $y_x^*=\frac{2^{t-1}(2i+1)}{2^t}+\frac{1}{2}=i+1$，由于 $i\in\mathbb{Z}$，故 $y_x^*\in\mathbb{Z}$，$u_x\leq y_x<y_x^*=\lfloor y_x^*\rfloor=u_x^*$，所以 $u_x<u_x^*$. 由于 $y_x^*-y_x=x\left(\frac{1}{2^t}-\frac{p}{q}\right)=\frac{x}{2^tq}<\frac{1}{2^t}<1$，因此 $y_x>y_x^*-1$，$u_x=\lfloor y_x\rfloor\geq\lfloor y_x^*-1\rfloor=\lfloor y_x^*\rfloor-1=u_x^*-1$. 由于 u_x、u_x^* 为整数，$u_x<u_x^*$ 且 $u_x\geq u_x^*-1$，所以 $u_x=u_x^*-1$，证毕.

②$x\notin\mathrm{W}$. 使 $y_x^*\in\mathbb{Z}$ 成立的输入 x 必须满足 $2^{t-1}(2i+1)$ 的形式，其中 $i\in\mathbb{Z}$，$0\leq i\leq p-1$，所以对于舍入算法的输入 $x\in\mathbb{Z}_q$，$x\notin\mathrm{W}$ 时 $y_x^*\notin\mathbb{Z}$.

(i) $u_x^*=0$ 时，由于 $x\in\mathbb{Z}_q$，即 $x\in[0,q-1]$，因此 $u_x=\left\lfloor\frac{p}{q}\cdot x+\frac{1}{2}\right\rfloor\geq 0$，即 $u_x\geq u_x^*=0$；

(ii) $u_x^*\geq 1$ 时，记 $x_i=2^{t-1}(2i+1)\in\mathrm{W}$，其中 $i\in\mathbb{Z}$，$0\leq i\leq p-1$. 由于 $y_{x_i}^*=i+1\in[1,p]$，因此对任意 $x\notin\mathrm{W}$，存在 $x_i\in\mathrm{W}$ 且 $x_i\leq x-1$ 使 $u_x^*=y_{x_i}^*$，因此 $y_x^*-u_x^*=y_x^*-y_{x_i}^*\geq y_x^*-y_{x-1}^*=\frac{1}{2^t}$，

$$y_x=y_x^*-\frac{1}{2^t}\cdot\frac{x}{q}=u_x^*+(y_x^*-u_x^*)-\frac{1}{2^t}\cdot\frac{x}{q}$$
$$\geq u_x^*+\frac{1}{2^t}-\frac{1}{2^t}\cdot\frac{x}{q}\geq u_x^*+\frac{1}{2^t}\left(1-\frac{x}{q}\right),$$

所以 $u_x=\lfloor y_x\rfloor\geq\left\lfloor u_x^*+\frac{1}{2^t}\left(1-\frac{x}{q}\right)\right\rfloor\geq u_x^*$，即 $u_x\geq u_x^*$.

综合(i)，(ii)有 $u_x\geq u_x^*$. 因为 $y_x<y_x^*$，所以有 $u_x=\lfloor y_x\rfloor\leq\lfloor y_x^*\rfloor=u_x^*$. 由于 u_x、u_x^* 为整数，$u_x\geq u_x^*$ 且 $u_x\leq u_x^*$，所以 $u_x=u_x^*$，证毕.

考虑一般情形，$q=p\cdot 2^t+r$，$1\leq r<p$. $r\leq 2^t$ 时，误差点集合为 $\{2^{t-1}(2i+1)+j\}_{0\leq i\leq p-1,0\leq j<\frac{r(i+0.5)}{p},i,j\in\mathbb{Z}}$，误差为 1，新舍入算法的推广(New General-RA，NG-RA)见算法 9，该算法的存储开销至少为 q 比特. $2^t<r<p$ 时，某些误差点的误差大于 1，此时算法需要的存储开销至少为 $\lceil rq/2^t\rceil$ 比特，误差点的计算较复杂，可遍历 $\mathbb{Z}_q$ 中所有元素，计算并存储每个点的误差.

算法 9.　NG-RA(x,p,q,t,r).

输入：自然数 x，正整数 p,q,t,r；$q=p\times 2^t+r$，$x,r<p$，$r<=2^t$

输出：$u=\lfloor (p/q)\cdot x\rceil \bmod p$

预计算

1. Pre$[q]=\{0\}$
2. FOR $i=0$ TO $p-1$ DO
3. 　FOR $j=0$ TO $r*(i+0.5)/p$ DO
4. 　　Pre$[(2i+1)\ll(t-1)+j]=1$

舍入计算

1. $u=\lfloor (x+2^{t-1})\gg t\rfloor-Pre[x]$
2. RETURN u

格密码方案的模数通常是预先选定的，实际中模数通常作为常量参与运算. 与模整数类似，GCC编译器会对 p,q 为常量的舍入计算进行优化：比值 q/p 为 2 的方幂时，利用移位加速舍入运算；否则，舍入实现方法类似于 Barrett 算法. 本文分别测试了变量和常量舍入计算的运行时间，见表 8，算法 D1 和算法 D2 见附录 D. 对于模数为常量的舍入计算，与传统舍入方法相比（预计算浮点数 p/q），新舍入算法效率提高了 11%左右；新舍入算法的效率略低于先加法再移位，二者效率相差不超过 2.5%.

表 8 舍入算法运行时间

数据类型	舍入算法	运行时间/μs		
		$N=256$	$N=512$	$N=1024$
变量	无优化	2.437	4.849	9.723
	算法 D1. 传统舍入方法	1.502	2.894	5.769
	算法 D2. 先加法再移位	0.542	1.061	2.091
	算法 8. 新舍入算法(本文)	0.583	1.154	2.403
常量	无优化	0.676	1.319	2.616
	算法 D1. 传统舍入方法	0.618	1.285	2.436
	算法 D2. 先加法再移位	0.533	1.066	2.120
	算法 8. 新舍入算法(本文)	0.548	1.087	2.166

本文提出的舍入计算初始化算法（RA_Setup）见算法 10，该算法输入参数 p 和 q，输出适用且高效的舍入算法名称. 具体地，比值 q/p 为 2 的方幂时，采用先加法再移位（Addition then Shift-RA，AS-RA）. 否则，$(q-1)/p$ 为 2 的方幂时，采用新舍入算法（New-RA）. 否则，采用传统舍入方法（Trivial-RA）.

算法 10. RA_Setup(p,q).

输入：正整数 p,q

输出：字符串 RA. 其含义为 RLWR/MLWR 计算阶段采用的舍入算法名称

1. $t=\lfloor \log(q/p) \rfloor$
2. IF $q==p*2^t$ THEN
3. RETURN "AS-RA"
4. ELSE IF $q==p*2^t+1$ THEN
5. RETURN "New-RA"
6. ELSE RETURN "Trivial-RA"

4.4 通用 RLWR 实现算法

本小节提出一种通用的 RLWR 实现算法，见算法 11，该算法的输入为 RLWR 参数 $N,f,p,q,S,$ P[$s_i\neq 0$]，CPU 字长 B，多项式 a 和 s；输出为 RLWR 分布中的 b. 通用 RLWR 实现算法分为初始化模块和 RLWR 计算模块两部分.

初始化模块根据 RLWR 参数和 CPU 字长，灵活选取高效的多项式乘法、模约化和舍入计算实现方案，并完成相关预计算，例如计算并存储 FFT 或 NTT 的旋转因子，或执行新舍入算法的预计算模块. RLWR 方案参数以及运行平台 CPU 字长确定后，初始化模块只需执行一次.

RLWR 计算模块用于计算 RLWR 分布中的 b. 初始化模块已完成了预计算，因此 RLWR 计算模块输入的变量仅为多项式 a 和 s.

算法 11. 通用 RLWR 实现算法.

输入：正整数 N,p,q,S,B，浮点数 P，系数向量 $\boldsymbol{f},\boldsymbol{a},\boldsymbol{s}$. 其中浮点数 P 即概率 P[$s_i\neq 0$]

输出：$b(x)$ 的系数向量 $\boldsymbol{b}$. $b(x)=\lfloor (p/q)\cdot a(x)\cdot s(x) \rceil \bmod p$

初始化

1. 调用多项式乘法初始化算法，得到字符串 $PMA=$ PMA_Setup($N,\boldsymbol{f},q,S,B,P$)，并完成 PMA 需要的预计算
2. 调用模约化初始化算法，得到字符串数组 $\boldsymbol{MRA}=$ MRA_Setup(q,PMA)
3. 调用舍入计算初始化算法，得到字符串 $RA=$ RA_Setup(p,q)，并完成 RA 的相关预计算

RLWR 计算

1. $\boldsymbol{b}_1=PMA(\boldsymbol{a},\boldsymbol{s})$ //多项式乘法
2. $\boldsymbol{b}_1=MRA[0](\boldsymbol{b}_1)$ //模多项式
3. FOR $k=0$ TO $N-1$ DO
4. $\boldsymbol{b}_2[k]=MRA[1](\boldsymbol{b}_1[k])$ //模整数
5. $\boldsymbol{b}[k]=RA(\boldsymbol{b}_2[k])$ //舍入计算
6. RETURN $\boldsymbol{b}$

4.5 通用 MLWR 实现算法

本小节提出一种通用的 MLWR 实现算法，见算法 12，该算法的输入为 MLWR 参数 $l,N,f,p,q,$ S，P[$s_i\neq 0$]，CPU 字长 B，多项式矩阵 $\boldsymbol{a}$ 和多项式向量 $\boldsymbol{s}$；输出为 MLWR 分布中的 $\boldsymbol{b}$. 该算法分为初始化模块和 MLWR 计算模块两部分. 初始化模块与通用 RLWR 实现算法中的初始化模块相同. MLWR 计算模块用于计算 MLWR 分布中的向量 $\boldsymbol{b}$.

MLWR 计算模块首先计算矩阵乘法

$$\boldsymbol{b}=\boldsymbol{a}\boldsymbol{s}=\begin{pmatrix} a_{11}s_1+a_{12}s_2+\cdots+a_{1l}s_l \\ \cdots \\ a_{l1}s_1+a_{l2}s_2+\cdots+a_{ll}s_l \end{pmatrix},$$

该矩阵乘法可分解为 $l\times l$ 次多项式乘法和 $l\times(l-1)$ 次多项式加法. 若多项式乘法直接采用 FFT 乘法算法或 NTT 类乘法算法，那么多项式 $s_1,s_2,\cdots,s_l$ 都需执行 l 次正向 FFT 或 NTT(-BAR)计算. 将 FFT 乘法算法或 NTT 类乘法算法分解为正向计算、点值乘法、逆向计算，并去掉重复的计算后，总计可省略 $l\times(l-1)$ 次正向 FFT 或 NTT(-BAR)计算. 考虑到多项式乘法后还需执行 $l-1$ 次系数表达的

多项式加法，例如，$\boldsymbol{b}$ 的第 i 个元素 $b_i=a_{i1}s_1+a_{i2}s_2+\cdots+a_{il}s_l$. 对于向量 $\boldsymbol{b}$ 的所有元素，若先执行点值表达的多项式加法，再执行逆向 FFT 或 NTT(-BAR)计算，总计可省略 $l\times(l-1)$ 次逆向 FFT 或 NTT(-BAR)计算. 注意，当逆向 NTT(-BAR)算法的 NTT 模数 M 不是方案模数 q 时，需保证 l 倍的乘积多项式的最大系数小于 NTT 模数，即需满足 $lqSN<2M$，否则不能省略逆向 NTT(-BAR)计算.

算法 12.　通用 MLWR 实现算法.

输入：正整数 l,N,p,q,S,B，浮点数 P，系数向量 $\boldsymbol{f},\boldsymbol{a},\boldsymbol{s}$. 其中浮点数 P 即概率 $\mathrm{P}[s_i\neq 0]$，$\boldsymbol{f}$ 存储于一维数组，$\boldsymbol{s}$ 存储于二维数组，$\boldsymbol{a}$ 存储于三维数组($\boldsymbol{a}[i][j]$为 a_{ij} 的系数向量)

输出：二维数组 $\boldsymbol{b}$. $\boldsymbol{b}=\lfloor(p/q)\cdot\boldsymbol{a}\cdot\boldsymbol{s}\rceil \bmod p$

初始化

1. 调用多项式乘法初始化算法，得到字符串 $PMA=$ PMA_Setup($N,\boldsymbol{f},q,S,B,P$)，并完成 PMA 需要的预计算
2. 调用模约化初始化算法，得到字符串数组 $\boldsymbol{MRA}=$ MRA_Setup(q,PMA)
3. 调用舍入计算初始化算法，得到字符串 $RA=$ RA_Setup(p,q)，并完成 RA 的相关预计算

MLWR 计算

1. $\boldsymbol{b}_1=\{0\}$
2. $n=2^{\lceil\log N\rceil+1}$
3. IF ($PMA==$"FFT-PMA") THEN
4. FOR $j=0$ TO $l-1$ DO
5. $\boldsymbol{s\_fft}[j]=\mathrm{FFT}(\boldsymbol{s}[j],n)$ //$\boldsymbol{s\_fft}$ 是二维数组
6. FOR $i=0$ TO $l-1$ DO
7. FOR $j=0$ TO $l-1$ DO
8. $\boldsymbol{a\_fft}[i][j]=\mathrm{FFT}(\boldsymbol{a}[i][j],n)$ //$\boldsymbol{a\_fft}$ 是三维数组
9. FOR $k=0$ TO $n-1$ DO
10. $\boldsymbol{b}_1[i][k]=\boldsymbol{b}_1[i][k]+\boldsymbol{a\_fft}[i][j][k]*\boldsymbol{s\_fft}[j][k]$
11. $\boldsymbol{b}_1[i]=\mathrm{IFFT}(\boldsymbol{b}_1[i],n)$
12. ELSE IF ($PMA==$"NTT-PMA_q,2^{k+1}") THEN
13. FOR $j=0$ TO $l-1$ DO
14. $\boldsymbol{s\_ntt}[j]=\mathrm{NTT}(\boldsymbol{s}[j],n,q,\alpha)$
15. FOR $i=0$ TO $l-1$ DO
16. FOR $j=0$ TO $l-1$ DO
17. $\boldsymbol{a\_ntt}[i][j]=\mathrm{NTT}(\boldsymbol{a}[i][j],n,q,\alpha)$
18. FOR $k=0$ TO $n-1$ DO
19. $\boldsymbol{b}_1[i][k]=(\boldsymbol{b}_1[i][k]+\boldsymbol{a\_ntt}[i][j][k]*\boldsymbol{s\_ntt}[j][k])\%q$
20. $\boldsymbol{b}_1[i]=\mathrm{INTT}(\boldsymbol{b}_1[i],n,q,\alpha)$
21. ELSE IF ($PMA==$"NTT-PMA_M,2^{k+1}") THEN
22. FOR $j=0$ TO $l-1$ DO
23. $\boldsymbol{s\_ntt}[j]=\mathrm{NTT}(\boldsymbol{s}[j],n,M,\alpha)$
24. FOR $i=0$ TO $l-1$ DO
25. FOR $j=0$ TO $l-1$ DO
26. $\boldsymbol{a\_ntt}[i][j]=\mathrm{NTT}(\boldsymbol{a}[i][j],n,M,\alpha)$
27. IF ($l*q*S*N<2*M$) THEN
28. FOR $i=0$ TO $l-1$ DO
29. FOR $j=0$ TO $l-1$ DO
30. FOR $k=0$ TO $n-1$ DO
31. $\boldsymbol{b}_1[i][k]=(\boldsymbol{b}_1[i][k]+\boldsymbol{a\_ntt}[i][j][k]*\boldsymbol{s\_ntt}[j][k])\%M$
32. $\boldsymbol{b}_1[i]=\mathrm{INTT}(\boldsymbol{b}_1[i],n,M,\alpha)$
33. ELSE
34. $\boldsymbol{b}_1\_\boldsymbol{temp}[l][l][n]=\{0\}$ //新建临时数组
35. FOR $i=0$ TO $l-1$ DO
36. FOR $j=0$ TO $l-1$ DO
37. FOR $k=0$ TO $n-1$ DO
38. $\boldsymbol{b}_1\_\boldsymbol{temp}[i][j][k]=(\boldsymbol{a\_ntt}[i][j][k]*\boldsymbol{s\_ntt}[j][k])\%M$
39. $\boldsymbol{b}_1\_\boldsymbol{temp}[i][j]=\mathrm{INTT}(\boldsymbol{b}_1\_\boldsymbol{temp}[i][j],n,M,\alpha)$
40. FOR $k=0$ TO $n-1$ DO
41. $\boldsymbol{b}_1[i][k]=(\boldsymbol{b}_1[i][k]+\boldsymbol{b}_1\_\boldsymbol{temp}[i][j][k])\%M$
42. ELSE IF ($PMA==$"NTT-NWC_q,N") THEN
43. FOR $j=0$ TO $l-1$ DO
44. $\boldsymbol{s\_ntt}[j]=\text{NTT-BAR}(\boldsymbol{s}[j],N,q,\varphi)$
45. FOR $i=0$ TO $l-1$ DO
46. FOR $j=0$ TO $l-1$ DO
47. $\boldsymbol{a\_ntt}[i][j]=\text{NTT-BAR}(\boldsymbol{a}[i][j],N,q,\varphi)$
48. FOR $k=0$ TO $N-1$ DO
49. $\boldsymbol{b}_1[i][k]=(\boldsymbol{b}_1[i][k]+\boldsymbol{a\_ntt}[i][j][k]*\boldsymbol{s\_ntt}[j][k])\%q$
50. $\boldsymbol{b}_1[i]=\text{INTT-BAR}(\boldsymbol{b}_1[i],N,q,\varphi)$
51. ELSE IF ($PMA==$"NTT-NWC_M,N") THEN
52. FOR $j=0$ TO $l-1$ DO
53. $\boldsymbol{s\_ntt}[j]=\text{NTT-BAR}(\boldsymbol{s}[j],N,M,\varphi)$
54. FOR $i=0$ TO $l-1$ DO
55. FOR $j=0$ TO $l-1$ DO
56. $\boldsymbol{a\_ntt}[i][j]=\text{NTT-BAR}(\boldsymbol{a}[i][j],N,M,\varphi)$
57. IF ($l*q*S*N<2*M$) THEN
58. FOR $i=0$ TO $l-1$ DO
59. FOR $j=0$ TO $l-1$ DO
60. FOR $k=0$ TO $N-1$ DO
61. $\boldsymbol{b}_1[i][k]=(\boldsymbol{b}_1[i][k]+\boldsymbol{a\_ntt}[i][j][k]*\boldsymbol{s\_ntt}[j][k])\%M$
62. $\boldsymbol{b}_1[i]=\text{INTT-BAR}(\boldsymbol{b}_1[i],N,M,\varphi)$
63. ELSE
64. $\boldsymbol{b}_1\_\boldsymbol{temp}[l][l][N]=\{0\}$ //新建临时数组
65. FOR $i=0$ TO $l-1$ DO

```
66.        FOR j=0 TO l-1 DO
67.          FOR k=0 TO n-1 DO
68.            b1_temp[i][j][k]=(a_ntt[i][j][k] *
                                 s_ntt[j][k])%M
69.          b1_temp[i][j]=INTT-BAR(b1_temp[i]
                                    [j],N,M,φ)
70.          FOR k=0 TO n-1 DO
71.            b1[i][k]=(b1[i][k]+b1_temp[i][j]
                         [k])%M
72. ELSE
73.   b1_temp[l][l][N]={0} //新建临时数组
74.   FOR i=0 TO l-1 DO
75.     FOR j=0 TO l-1 DO
76.       b1_temp[i][j]=PMA(a[i][j],s[j])
77.       FOR k=0 TO n-1 DO
78.         b1[i][k]=b1[i][k]+b1_temp[i][j][k]
79. FOR i=0 TO l-1 DO
80.   b1[i]=MRA[0](b1[i])
81.   FOR k=0 TO N-1 DO
82.     b2[i][k]=MRA[1](b1[i][k])
83.     b[i][k]=RA(b2[i][k])
84. RETURN b
```

5 RLWR 和 MLWR 参数选取方法

RLWR 和 MLWR 软件实现效率均受参数影响显著，本节根据不同参数下 RLWR 和 MLWR 的效率，提出了 RLWR 和 MLWR 快速实现参数选取方法. 相同运行平台、相同安全等级下，合理的参数选择可以提升 RLWR 和 MLWR 的实现效率.

5.1 RLWR 快速实现参数选取方法

RLWR 实现方法由参数 $N,f,p,q,S,\mathrm{P}[s_i\neq 0]$ 决定，本小节提出一种 RLWR 参数选取方法. 注意，稀疏乘法算法易受侧信道攻击，由于 RLWR 分布中的多项式 s 通常是秘密信息，为保障算法的安全性，应尽量避免采用稀疏乘法算法，因此本小节提出的 RLWR 参数选取方法不单独考虑 $S<4$ 的情形.

参数 f 主要影响多项式乘法和模多项式的效率. $f(x)=x^N\pm 1$ 是采用 NTT 负/正折叠卷积的必要条件；通用模多项式是采用减法和乘法实现的，f 的系数取 2 的方幂或 ± 1 可加速乘法操作，f 的部分系数取 0 可直接省略部分减法及乘法操作. 因此，f 优先取 $x^N\pm 1$，否则 f 的系数优先取 2 的方幂、0 或 ± 1.

参数 N 影响 RLWR 的安全等级，此外主要影响多项式乘法的效率. $N=2^{\lceil\log N\rceil}$ 是采用 NTT 正/负折叠卷积的必要条件；另外，N 决定了 NTT 和 FFT 的输入规模 n，通常取 $n=2^{\lceil\log N\rceil+1}$. 因此，$N$ 优先取 2 的方幂，否则优先取 $\lceil\log N\rceil-\log N$ 较小的 N，本文假设安全等级由 $\lfloor\log N\rceil$ 决定，因此应优先取满足 $\lceil\log N\rceil-\log N<0.5$ 的 N，实际中应结合具体安全等级确定 $\lceil\log N\rceil-\log N$ 的边界值.

参数 q 和 S 主要影响多项式乘法的效率，本文将参数 q 和 S 分为以下三类：

(A) $q\in \mathrm{M}(n,B)$，S 为任意正整数.

(B) q 不满足条件(A)，但 $qSN<2\mathrm{M}_{\max}(n,B)$.

(C) 参数 q 和 S 不满足条件(A)和(B).

其中，B 是运行平台的 CPU 字长，$f(x)=x^N\pm 1$ 且 $N=2^{\lceil\log N\rceil}$ 时，$n=N$，否则 $n=2^{\lceil\log N\rceil+1}$；$\mathrm{M}(N,B)$，$\mathrm{M}_{\max}(N,B)$ 的定义见 4.1 节. 条件(A)或(B)是 NTT 类乘法算法的必要条件. 参数 q 也会影响模整数的效率：q 满足条件(A)时可省略模整数运算，q 取 2 的方幂时可采用按位与操作加速模整数运算. 由于模整数运算的耗时远小于多项式乘法，其他实现方案相同时，采用不同模整数实现方案的 RLWR 运行时间相差不超过 3%，本文的 RLWR 参数选取方法暂时忽略 q 对模整数效率的影响. 因此，参数 q 和 S 优先取满足条件(A)的值，否则优先取满足条件(B)的值.

参数 p 的取值主要影响舍入计算的效率. 参数 q 确定后，参数 $p=q/2^{\lfloor\log(q/p)\rfloor}$ 时，可采用移位操作加速舍入计算；参数 $p=(q-1)/2^{\lfloor\log(q/p)\rfloor}$ 时，可采用新舍入算法. 因此，p 优先取 $q/2^{\lfloor\log(q/p)\rfloor}$，否则优先取 $(q-1)/2^{\lfloor\log(q/p)\rfloor}$. 舍入计算的耗时同样远小于多项式乘法，参数 p 取不同值时，通用 RLWR 实现算的效率相差不过 3%.

参数 $\mathrm{P}[s_i\neq 0]$ 主要影响稀疏乘法算法的效率. $\mathrm{P}[s_i\neq 0]$ 越小稀疏乘法的效率越高，故若采用稀疏乘法算法，则优先取 $\mathrm{P}[s_i\neq 0]<1/2^{k-5}$，否则优先取 $\mathrm{P}[s_i\neq 0]<2^{k+6}/N^2$，否则优先取 $\mathrm{P}[s_i\neq 0]<2^{k+7}/N^2$，其中 $k=\lceil\log N\rceil$.

本文根据 RLWR 实现效率，将相同安全等级下，RLWR 常用参数分为 37 类并给出各类 RLWR 参数的优先级，见表 9，优先级一栏中的数值越小，表示该类参数的实现效率越高. 表 9 中给出了 RLWR 参考效率，可以看出，对于每一类参数，都至少存在其他两类与其效率相近的参数，例如优先级 1～5 这 5 类参数的 RLWR 效率相差不超过 5%，因此，若某类参数存在安全风险，那么可在不大幅降低 RLWR 实现效率的前提下，采用其他参数作为替换.

表 9　RLWR 参数优先级

f	$\lceil\log N\rceil-\log N$	q,S	p	优先级	RLWR 参考效率（$\lfloor\log N\rceil=9$，单位：μs）
$x^N\pm1$	0	A	$(q-1)/p=2^t$	1	46
			其他	2	
		B	$q/p=2^t$	3	
			$(q-1)/p=2^t$	4	
			其他	5	
	(0,0.5)	A	$(q-1)/p=2^t$	6	100
			其他	7	
		B	$q/p=2^t$	8	
			$(q-1)/p=2^t$	9	
			其他	10	
	[0,0.5)	C	$q/p=2^t$	11	200
			$(q-1)/p=2^t$	12	
			其他	13	
	[0.5,1)	A	$(q-1)/p=2^t$	14	
			其他	15	
		B	$q/p=2^t$	16	
			$(q-1)/p=2^t$	17	
			其他	18	
	[0.5,1)	C	$q/p=2^t$	19	430
			$(q-1)/p=2^t$	20	
			其他	21	
其他	[0,0.5)	A	$(q-1)/p=2^t$	22	890
			其他	23	
		B	$q/p=2^t$	24	
			$(q-1)/p=2^t$	25	
			其他	26	
	[0,0.5)	C	$q/p=2^t$	27	990
			$(q-1)/p=2^t$	28	
			其他	29	
	[0.5,1)	A	$(q-1)/p=2^t$	30	
			其他	31	
		B	$q/p=2^t$	32	
			$(q-1)/p=2^t$	33	
			其他	34	
	[0.5,1)	C	$q/p=2^t$	35	1220
			$(q-1)/p=2^t$	36	
			其他	37	

注：$t=\lfloor\log(q/p)\rfloor$. 表中给出的 RLWR 效率为参考值，实际效率见 6.2 节表 14，参考效率与实际效率相差不超过 5%.

5.2　MLWR 快速实现参数选取方法

MLWR 实现方法由参数 $N,f,p,q,l,S,\mathrm{P}[s_i\neq0]$ 决定，其中，参数 $N,f,p,q,S,\mathrm{P}[s_i\neq0]$ 的选取方法与 5.1 节中 RLWR 对应的参数选取方法相同.

参数 l 影响 MLWR 的安全等级，此外主要影响矩阵乘法的效率. 对于采用 NTT 类乘法算法的 MLWR，当 NTT 或 NTT-BAR 算法选取的 NTT 模数 M 不是方案模数 q 时，若参数 l 满足 $lqSN<2M$，则可省略 $l\times(l-1)$ 次逆向 NTT 或 NTT-BAR 计算. 故上述情况应优先选取 l 满足 $l<2\mathrm{M}_{\max}(n,B)/qSN$，其中 n 表示对应的 NTT 输入规模.

$l=1$ 时，MLWR 等价于 RLWR，此时 MLWR 实现效率及各类 MLWR 参数的优先级可参考表 9. $l>1$ 时，MLWR 参数的优先级见表 10. 考虑到参数 p 的取值主要影响舍入计算的效率，而舍入计算的耗时远小于多项式乘法，故表 10 暂时忽略参数 p.

表 10　MLWR 参数优先级

f	$\lceil\log N\rceil-\log N$	q,S	l	优先级
$x^N\pm1$	0	A	/	1
		B	$l<2\mathrm{M}_{\max}(n,B)/qSN$	
		B	$l>2\mathrm{M}_{\max}(n,B)/qSN$	2
	(0,0.5)	A	/	3
		B	$l<2\mathrm{M}_{\max}(n,B)/qSN$	
		B	$l>2\mathrm{M}_{\max}(n,B)/qSN$	4
	[0,0.5)	C	/	5
	[0.5,1)	A	/	
		B	$l<2\mathrm{M}_{\max}(n,B)/qSN$	
		B	$l>2\mathrm{M}_{\max}(n,B)/qSN$	6
	[0.5,1)	C	/	7
其他	[0,0.5)	A	/	8
		B	$l<2\mathrm{M}_{\max}(n,B)/qSN$	
		B	$l>2\mathrm{M}_{\max}(n,B)/qSN$	9
	[0,0.5)	C	/	
	[0.5,1)	A	/	10
		B	$l<2\mathrm{M}_{\max}(n,B)/qSN$	
		B	$l>2\mathrm{M}_{\max}(n,B)/qSN$	11
	[0.5,1)	C	/	12

注：参数 q,S 类别 ABC 的定义见 5.1 节，n 表示该类 MLWR 参数下对应的 NTT 类乘法算法的输入规模，/表示该参数可取任意值.

6　软件实现与效率分析

本节在 64 位 Intel(R) Core(TM) i7-6700 CPU@3.40 GHz 平台，ubuntu16.04 操作系统上，采用 GCC 编译器测试了相关算法的实际运行时间，并将本文的工作与第 3 节中的相关工作进行了对比分析. 如不作特殊说明，本节中默认未使用编译优化指令.

6.1　NTT 乘法优化实现方法及其效率分析

本文发现，GCC 编译器会优化常量的模运算，在不直接使用汇编的情况下，采用 Montgomery 等模整数优化算法反而会产生额外的存取负担，故无优化的模运算（例如 C 语言中%运算符）的效率高于 Montgomery 等优化算法. 虽然计算单次模整数的效率差异并不显著，但在需要执行大量模运算的 NTT 中，采用%运算符可以显著提高 NTT 的效率. 本文将%运算符应用于 Dilithum 方案实现，见表 11，与采用 Montgomery 算法相比，采用%运算符后 NTT 负折叠卷积的效率最多可提高 15%左右；此外，数据类型采用无符号整数可以进一步提升模整数的效率，与采用无符号整数的 Montgomery 算法相比，采用%运算符以及无符号整数后 NTT 负折叠卷积的效率最多可提升约 17.2%.

表 11 Dilithum 实现中 NTT 负折叠卷积运行时间

模整数方法	整数类型	NTT 负折叠卷积运行时间/μs		
		$N=256$	$N=512$	$N=1024$
Montgomery(Dilithum)	无符号	33.6	67.8	140.0
%运算符(本文)	有符号	28.3	59.4	135.0
%运算符(本文)	无符号	27.8	57.5	122.6

注：Dilithum 并未直接提供输入规模为 512 和 1024 的 NTT 实现，本文在生成相应的单位根方幂后使用 Dilithum 原始代码进行效率测试.

此外，本文测试了 Dilithum、Kyber、qTESLA 方案现有 NTT 负折叠卷积实现，以及采用%运算符的 NTT 负折叠卷积运行时间，见表 12，可以看出，采用%运算符以及无符号整数能显著提升 NTT 负折叠卷积的效率，在选择了合适模数的基础上，与现有实现相比 NTT 负折叠卷积的效率最多可以提升约 30%.

表 12 NTT 负折叠卷积运行时间

NTT实现及模数	整数类型	模整数方法	运行时间/μs		
			$N=256$	$N=512$	$N=1024$
Dilithum $1023\times2^{13}+1$	无符号	Montgomery	33.6	/	/
Kyber $13\times2^{8}+1$	有符号	Montgomery	35.2	/	/
qTESLA_I $4107\times2^{10}+1$	有符号	Montgomery	/	56.4	/
qTESLA_III $513\times2^{14}+1$	有符号	Montgomery	/	/	133.2
本文 $1023\times2^{13}+1$	无符号	%运算符	27.8	57.5	122.6
本文 $119\times2^{23}+1$	无符号	%运算符	20.4	43.7	92.6

另一方面，对于 B 位 CPU 平台，只要 NTT 模数小于 $2^{B/2}$，NTT 的实现就无需引入大整数运算，此时 NTT 的效率与 NTT 模数的大小并不相关. NTT 主要由加法、减法、乘法以及模运算组成，只要计算结果在 CPU 字长内，那么执行加法、减法、乘法运算的时间通常是固定的. 本文发现，模运算采用%运算符后，NTT 效率存在差异的本质原因在于模数的结构不同. 本文在 64 位 CPU 平台测试了满足 $M<2^{32}$ 的 3 个不同模数 NTT 负折叠卷积的运行时间，见表 13，它们的效率最多相差 25%；此外，本文实现了 64 位模数的 NTT 负折叠卷积，由于涉及到大整数运算，其效率远低于 32 位模数的 NTT 负折叠卷积. 如何结合现有编译器的模运算实现方法选择合适的 NTT 模数还有待研究.

表 13 不同模数下 NTT 负折叠卷积运行时间

NTT 模数	位数	NTT 负折叠卷积运行时间/μs		
		$N=256$	$N=512$	$N=1024$
$119\times2^{23}+1$	30 位	20.4	43.7	92.6
$2^{16}+1$	17 位	22.9	49.4	102.9
$1023\times2^{13}+1$(Dilithum 模数)	23 位	27.8	57.5	122.6
$(3\times2^{30}+1)^2$	64 位	186.2	409.1	893.8

注：数据类型为无符号整数，NTT 内部采用%运算符计算模整数.

6.2 通用 RLWR 实现算法及其效率

本文测试了环 $\mathbb{Z}_q[x]/(x^N+1)$ 上，$S\geq4$ 时，N 分别取 511、512 及 513 的 RLWR 运行时间，见表 14，其中，多项式乘法的运行时间占 RLWR 整体的 97%以上，多项式乘法的效率决定了 RLWR 效率.

表 14 不同参数 N,p,q 下通用 RLWR 实现算法运行时间

N	q,S	p	多项式乘法及参考运行时间/μs	模约化及运行时间/μs		舍入计算及运行时间/μs	RLWR 参考运行时间/μs
				模多项式	模整数		
511	A	$(q-1)/p=2^t$	NTT($n=1024$,模数 q),97.1	减法,1.4	无	新舍入算法,1.1	99.6
	A	其他	NTT($n=1024$,模数 q),97.1	减法,1.4	无	预计算 p/q,1.3	99.8
	B	$q/p=2^t$	NTT($n=1024$,模数 M),97.1	减法,1.4	%运算符,1.3	先加法再移位,1.1	100.9
	B	$(q-1)/p=2^t$	NTT($n=1024$,模数 M),97.1	减法,1.4	%运算符,1.3	新舍入算法,1.1	100.9
	B	其他	NTT($n=1024$,模数 M),97.1	减法,1.4	%运算符,1.3	预计算 p/q,1.3	101.1
	C	$q/p=2^t$	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	先加法再移位,1.1	198.1
	C	$(q-1)/p=2^t$	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	新舍入算法,1.1	198.1
	C	其他	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	预计算 p/q,1.3	198.3
512	A	$(q-1)/p=2^t$	NTT 负折叠卷积($n=512$,模数 q),43.7	无	无	新舍入算法,1.1	44.8
	A	其他	NTT 负折叠卷积($n=512$,模数 q),43.7	无	无	预计算 p/q,1.3	45.0
	B	$q/p=2^t$	NTT 负折叠卷积($n=512$,模数 M),43.7	无	%运算符,1.3	先加法再移位,1.1	46.1
	B	$(q-1)/p=2^t$	NTT 负折叠卷积($n=512$,模数 M),43.7	无	%运算符,1.3	新舍入算法,1.1	46.1
	B	其他	NTT 负折叠卷积($n=512$,模数 M),43.7	无	%运算符,1.3	预计算 p/q,1.3	46.3
	C	$q/p=2^t$	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	先加法再移位,1.1	198.1
	C	$(q-1)/p=2^t$	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	新舍入算法,1.1	198.1
	C	其他	FFT($n=1024$),194.3	减法,1.4	%运算符,1.3	预计算 p/q,1.3	198.3

（续　表）

N	q,S	p	多项式乘法及参考运行时间/μs	模约化及运行时间/μs		入计算及运行时间/μs	RLWR 参考运行时间/μs
				模多项式	模整数		
513	A	$(q-1)/p=2^t$	NTT($n=2048$,模数 q),198.1	减法,1.4	无	新舍入算法,1.1	200.6
	A	其他	NTT($n=2048$,模数 q),198.1	减法,1.4	无	预计算 p/q,1.3	200.8
	B	$q/p=2^t$	NTT($n=2048$,模数 M),198.1	减法,1.4	%运算符,1.3	先加法再移位,1.1	201.9
	B	$(q-1)/p=2^t$	NTT($n=2048$,模数 M),198.1	减法,1.4	%运算符,1.3	新舍入算法,1.1	201.9
	B	其他	NTT($n=2048$,模数 M),198.1	减法,1.4	%运算符,1.3	预计算 p/q,1.3	202.1
	C	$q/p=2^t$	FFT($n=2048$),426.8	减法,1.4	%运算符,1.3	先加法再移位,1.1	430.6
	C	$(q-1)/p=2^t$	FFT($n=2048$),426.8	减法,1.4	%运算符,1.3	新舍入算法,1.1	430.6
	C	其他	FFT($n=2048$),426.8	减法,1.4	%运算符,1.3	预计算 p/q,1.3	430.8

注：N,p,q,S 均作为常量参与运算. t 表示任意正整数，n 表示算法的输入规模；M 可取任意满足 $qSN<2M$ 且 $M\in M(n,B)$ 的正整数，n 即输入规模，B 是 CPU 字长. 参数 q,S 类别 ABC 的定义见 5.1 节，对于 A 类参数，本文测试了 $q=119\times2^{23}+1$ 时 RLWR 及其基础操作的运行时间；对于 B 类参数，本文随机选取了 100 组参数 q，测试了平均运行时间，其中 NTT 模数取 $M=119\times2^{23}+1$；对于 C 类参数，本文随机选取了 100 组参数 q，测试了平均运行时间. 考虑到 NTT 模数可能会影响 NTT 乘法算法的效率，本文统一了 A 类和 B 类参数中的 NTT 模数，故 A 类参数仅参考了 $q=119\times2^{23}+1$ 的运行时间.

6.3　通用 MLWR 实现算法的应用及其效率分析

本文提出的通用 MLWR 实现算法可应用于 Saber 方案实现. Saber 方案中环为 $\mathbb{Z}_q[x]/(x^N+1)$，s 取自中心二项分布，$N=256$，$q=2^{13}$，$p=2^{10}$，$S=16$. Saber 提供三个参数集：Lightsaber、Saber 和 Firesaber，其中参数 l 分别取 2，3 和 4. Saber 方案的多项式乘法可采用 NTT 负折叠卷积（输入规模为 256，NTT 模数取 $119\times2^{23}+1$），同时可省略部分正向 NTT 和逆向 NTT 计算，无需模多项式操作，模整数采用按位与，舍入计算采用先加法再移位. 与本文方法的区别在于，Saber 团队第三轮提交的方案实现中，多项式乘法采用了 Toom-Cook (4way)和 Karatsuba 算法，模多项式采用了减法，因此本文本质上优化了多项式乘法，同时省略了模多项式操作. 在 64 位通用 CPU 平台，不使用编译器优化指令时（默认为 O0），与原有 Saber 第三轮实现相比，本文优化后的 Saber 密钥封装实现效率提升了 52%左右，见表 15.

表 15　Saber 方案运行时间

方案		运行时间/μs		NIST 安全级别
		Saber 第三轮实现	本文实现	
Light Saber ($l=2$)	密钥生成	226	121	1
	密钥封装	325	184	
	密钥解封装	409	226	
Saber ($l=3$)	密钥生成	480	210	3
	密钥封装	628	301	
	密钥解封装	756	362	
Fire Saber ($l=4$)	密钥生成	817	322	5
	密钥封装	1008	439	
	密钥解封装	1177	525	

本文测试了使用编译器优化指令的情况下 Saber($l=3$)方案的实现效率，见表 16. 采用 O/O1 优化指令的情况下，本方法优化后的 Saber 密钥封装实现效率提升了 28%左右，采用 O2 优化指令的 Saber 密钥封装实现效率提升了 31%左右. 但采用 O3 优化指令后，本文的方法未能提升实现效率，如何结合 O3 或其他编译器优化指令进一步优化 NTT 类乘法算法的软件实现还有待研究.

表 16　不同优化指令下 Saber 方案运行时间

优化指令	Saber($l=3$)	运行时间/μs	
		Saber 第三轮实现	本文实现
O/O1	密钥生成	132	83
	密钥封装	170	122
	密钥解封装	202	150
O2	密钥生成	126	76
	密钥封装	162	110
	密钥解封装	192	133
O3	密钥生成	45	70
	密钥封装	55	98
	密钥解封装	61	117

7　总结与展望

本文提出了通用高效的 RLWR 实现算法和 MLWR 实现算法，以及 RLWR 和 MLWR 快速实现参数选取方法，并在 64 位通用 CPU 平台完成了相关实验测试. 本文提出的通用 MLWR 实现算法可以应用于 Saber 方案实现，与 Saber 方案第三轮原始实现相比，使用 O0、O1、O2 优化指令时，采用本文方法后 Saber 密钥封装的效率分别提升了 52%、28%、31%；使用 O3 优化指令时，本文的方法未能提升 Saber 密钥封装效率，如何结合 O3 或其他优化指令提升通用 RLWR/MLWR 实现算法效率还有待研究. 此外，虽然本文是在 Intel 平台上完成了 RLWR 和 MLWR 实现方案的对比分析，但本文的结果对 ARM、AMD 等其他平台同样具有指导意义.

为提升多项式乘法实现效率，本文结合格密码常用的特殊参数，讨论了 NTT 类乘法算法的使用条件，给出了 NTT、NTT 负折叠卷积的限制条件与方案参数和 CPU 字长的量化关系，扩展了现有可快速实现的基于多项式环的密码方案参数空间. 除 RLWR 和 MLWR 外，该成果还可以应用于 Ring-LWE、NTRU 等密码方案的多项式乘法实现. 虽然 NTT 和 FFT 是目前实用的多项式乘法快速实现中理论渐近复杂度最低的算法，但是对于输入多项式的次数界 N 较小的情况，Karatsuba 或 Toom-Cook 乘法算法的实际效率可能优于 NTT 或 FFT 乘法算法. 结合格密码的参数选取特点，对非 FFT 和 NTT 乘法算法，特别是 Karatsuba 和 Toom-Cook 乘法算法的优化还有较大探索空间.

参 考 文 献

[1] Shor P W. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer//Proceedings of the 1st International Symposium on Algorithmic Number Theory. Ithaca, USA, 1994: 289

[2] Banerjee A, Peikert C, Rosen A. Pseudorandom functions and lattices//Proceedings of the 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques. Cambridge, UK, 2012: 719-737

[3] Alwen J, Krenn S, Pietrzak K, et al. Learning with rounding, revisited: New reduction, properties and applications//Proceedings of the 33rd Annual International Cryptology Conference. Santa Barbara, USA, 2013: 57-74

[4] Bogdanov A, Guo S, Masny D, et al. On the hardness of learning with rounding over small modulus//Proceedings of the 13th Theory of Cryptography Conference. Tel Aviv, Israel, 2016: 209-224

[5] Banerjee A, Peikert C. New and improved key-homomorphic pseudorandom functions//Proceedings of the 34th Annual International Cryptology Conference. Santa Barbara, USA, 2014: 353-370

[6] Xie X, Xue R, Zhang R. Deterministic public key encryption and identity based encryption from lattices in the auxiliary-input setting//Proceedings of the 8th International Conference on Security and Cryptography for Networks. Amalfi, Italy, 2012: 1-18

[7] Cheon J H, Park S, Lee J, et al. Lizard. Submission to the NIST Post-Quantum Cryptography Standardization, round1, 2017. Available at https://csrc. nist. gov/Projects/post-quantum-cryptography/Round-1-Submissions

[8] Garcia-Morchon O, Zhang Z, Bhattacharya S, et al. Round5 (Merger of HILA5 and Round2). Submission to the NIST Post-Quantum Cryptography Standardization, round2, 2019. Available at https://csrc. nist. gov/Projects/post-quantum-cryptography /round-2-submissions

[9] D'Anvers J, Karmakar A, Roy S S, et al. SABER. Submission to the NIST Post-Quantum Cryptography Standardization, round3, 2020. Available at https://csrc. nist. gov/Projects/post-quantum -cryptography/round-3-submissions

[10] Luo F, Wang F, Wang K, et al. LWR-based fully homomorphic encryption, revisited. Security and Communication Networks, 2018, 5967635: 1-12

[11] Karatsuba A, Ofman Y. Multiplication of many-digital numbers by automatic computers. Doklady Akademii nauk SSSR, 1962, 145(2): 293-294

[12] Toom A L. The complexity of a scheme of functional elements realizing the multiplication of integers. Doklady Akademii nauk SSSR, 1963, 3(3): 496-498

[13] Cook S A, Aanderaa S O. On the minimum computation time of functions. Transactions of the American Mathematical Society, 1969, 142: 291-314

[14] Cooley J W, Tukey J W. An algorithm for the machine computation of complex Fourier series. Mathematics of Computation, 1965, 19(90): 297-301

[15] Pollard J M. The fast Fourier transform in a finite field. Mathematics of Computation, 1971, 25(114): 365-374

[16] Nejatollahi H, Dutt N D, Ray S, et al. Post-quantum lattice-based cryptography implementations: A survey. ACM Computing Surveys, 2019, 51(6): 129:1-129:41

[17] Schwabe P, Avanzi R, Kiltz E, et al. CRYSTALS-KYBER. Submission to the NIST Post-Quantum Cryptography Standardization, round3, 2020. Available at https://csrc. nist. gov/Projects/post-quantum-cryptography/round-3-submissions

[18] Lyubashevsky V, Ducas L, Kiltz E, et al. CRYSTALS-DILITHIUM. Submission to the NIST Post-Quantum Cryptography Standardization, round3, 2020. Available at https://csrc. nist. gov/Projects/post-quantum-cryptography/round-3-submissions

[19] Bindel N, Akleylek S, Alkim E, et al. qTESLA. Submission to the NIST Post-Quantum Cryptography Standardization, round2, 2019. Available at https://csrc. nist. gov/Projects/post-quantum- cryptography/round-2-submissions

[20] Smart N P, Albrecht M R, Lindell Y, et al. LIMA. Submission to the NIST Post-Quantum Cryptography Standardization, round1, 2017. Available at https://csrc. nist. gov/Projects/post-quantum- cryptography/Round-1-Submissions

[21] Chung C M, Hwang V, Kannwischer M J, et al. NTT multiplication for NTT-unfriendly rings new speed records for Saber and NTRU on Cortex-M4 and AVX2. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, 2021(2): 159-188

[22] Montgomery P L. Modular multiplication without trial division. Mathematics of Computation, 1985, 44(170): 519-521

[23] Barret P. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor//Proceedings of the 6th Annual International Cryptology Conference. Santa Barbara, USA, 1986: 311-323

[24] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction

to Algorithms. Third Edition. Beijing: China Machine Press, 2012: 530(in Chinese)
(Cormen T H, Leiserson C E, Rivest R L 等. 算法导论. 第3版. 北京: 机械工业出版社, 2012: 530)

[25] Pöppelmann T, Güneysu T. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware// Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America. Santiago, Chile, 2012: 139-158

[26] Comba P G. Exponentiation cryptosystems on the IBM PC. IBM Systems Journal, 1990, 29(4): 526-538

[27] Schönhage A, Strassen V. Schnelle multiplikation großer Zahlen. Computing, 1971, 7(3-4): 281-292

[28] Fürer M. Faster integer multiplication. SIAM Journal on Computing, 2009, 39(3): 979-1005

[29] Harvey D, Hoeven J. Faster polynomial multiplication over finite fields using cyclotomic coefficient rings. Journal of Complexity, 2019, 54(OCT.): 101404.1-101404.18

[30] Espitau T, Fouque P, Gérard B, et al. Side-channel attacks on BLISS lattice-based signatures: exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers//Proceedings of the 24th Conference on Computer and Communications Security. Dallas, USA, 2017: 1857-1874

附录 A.　多项式乘法实现算法.

通过比特逆转操作,将多项式系数按照其在 FFT/NTT 中调用的顺序重新排列,即可实现迭代 FFT/NTT,比特逆转算法见算法 A1. 迭代 FFT 算法和迭代 NTT 算法分别见算法 A2 和算法 A3. NTT 负折叠卷积中调用的迭代 NTT-BAR 算法和迭代 INTT-BAR 算法描述分别见算法 A4 和算法 A5.

算法 A1.　Bitreverse($\boldsymbol{a}$).

输入:向量 $\boldsymbol{a}$. $\boldsymbol{a}$ 的长度为 2 的方幂

输出:向量 $\boldsymbol{a}$

1. $n=\boldsymbol{a}$.length　　//n 为 2 的方幂
2. $bitrev[n]=\{0\}$ //初始化整数数组 bitrev
3. FOR $i=0$ TO $n-1$ DO　//生成比特逆转数组
4. 　$temp\_i=i$
5. 　FOR $j=1$ TO log$n$ DO
6. 　　$temp=temp\_i\%2$;
7. 　　$bitrev[i]=(bitrev[i]\ll 1)+temp$
8. 　　$temp\_i=temp\_i\gg 1$
9. FOR $i=0$ TO $n-1$ DO
10. 　IF $i<bitrev[i]$ THEN
11. 　　$t=a[i]$
12. 　　$a[i]=a[bitrev[i]]$
13. 　　$a[bitrev[i]]=t$
14. RETURN $\boldsymbol{a}$

算法 A2.　FFT($\boldsymbol{a}$,n).

输入:向量 $\boldsymbol{a}$,正整数 n. n 为 2 的方幂

输出:向量 $\boldsymbol{A}$. $\boldsymbol{A}$ 为系数向量 $\boldsymbol{a}$ 对应的点值向量

1. 将 $\boldsymbol{a}$ 的长度扩展至 n,高位补 0
2. Bitreverse($\boldsymbol{a}$)
3. FOR $s=1$ TO log$n$ DO
4. 　　$m=2^s$
5. 　　$w_m=e^{2\pi i/m}$　//IFFT 中 w_m 取 $e^{2\pi i/m}$ 的共轭,完成预计算后可省略,i 是虚数单位
6. 　FOR $k=0$ TO $n-1$ BY m DO
7. 　　$w=1$　　//完成预计算后可省略
8. 　　FOR $j=0$ TO $m/2-1$ DO
9. 　　　$u=a[k+j]$
10. 　　　$v=w*a[k+j+m/2]$ //可预计算 w,并按调用顺序存储在数组中
11. 　　　$a[k+j]=u+v$
12. 　　　$a[k+j+m/2]=u-v$
13. 　　　$w=w*w_m$　//完成预计算后可省略
14. //IFFT 需要对所有 $a[i]$计算 $a[i]=a[i]/n$
15. RETURN $\boldsymbol{a}$

算法 A3.　NTT($\boldsymbol{a}$,n,M,α).

输入:向量 $\boldsymbol{a}$,正整数 n,M,α. n 为 2 的方幂,M 是规模为 n 的 NTT 模数,α 是 n 次单位根

输出:向量 $\boldsymbol{A}$. $\boldsymbol{A}$ 为系数向量 $\boldsymbol{a}$ 对应的点值向量

1. Bitreverse($\boldsymbol{a}$)
2. FOR $s=1$ TO log$n$ DO
3. 　$m=2^s$
4. 　$z_m=\alpha^{n/m}$　//INTT 中 z_m 取 $\alpha^{-n/m}$,完成预计算后该语句可省略
5. 　FOR $k=0$ TO $n-1$ BY m DO
6. 　　$z=1$　//完成预计算后该语句可省略
7. 　　FOR $j=0$ TO $m/2-1$ DO
8. 　　　$u=a[k+j]$
9. 　　　$v=z*a[k+j+m/2]$ //可预计算 z,并按调用顺序存储在数组中
10. 　　　$a[k+j]=(u+v)\%M$
11. 　　　$a[k+j+m/2]=(u-v)\%M$
12. 　　　$z=z*z_m$　//完成预计算后该语句可省略
13. //INTT 需要对所有$a[i]$计算$a[i]=(n^{-1}*a[i])\%M$
14. RETURN $\boldsymbol{a}$

算法 A4.　NTT-BAR($\boldsymbol{a}$,n,M,φ).

输入:向量 $\boldsymbol{a}$,正整数 n,M,φ. n 为 2 的方幂,M 是规模为 n 的 NTT 模数,φ 是 $2n$ 次单位根

输出:向量 $\bar{\boldsymbol{A}}=\mathrm{NTT}_{\alpha}(\bar{\boldsymbol{a}})$

1. Bitreverse($\boldsymbol{a}$)
2. $m=2$
3. WHILE $m<=n$ DO
4. 　$\psi_m=\varphi^{n/m}$　//完成预计算后该语句可省略
5. 　$z=\psi_m$　//完成预计算后该语句可省略

```
6.    FOR j=0 TO m/2−1 DO
7.      FOR k=0 TO n−1 BY m DO
8.        u=a[k+j]
9.        v=z*a[k+j+m/2] //可预计算 z,并按调用顺序存储在数组中
10.       a[k+j]=(u+v)%M
11.       a[k+j+m/2]=(u−v)%M
12.     z=(z*ψ_m*ψ_m)%M //完成预计算后可省略
13.   m=m*2
14. RETURN a
```

算法 A5. INTT-BAR($\boldsymbol{A}$,n,n^{-1},M,φ^{-1}).

输入:向量 $\boldsymbol{A}$,正整数 n,n^{-1},M,φ^{-1}. n 为 2 的方幂,M 是规模为 n 的 NTT 模数,φ 是 $2n$ 次单位根,n^{-1} 是 n 模 M 的逆

输出:向量 $\boldsymbol{a}$. $\boldsymbol{a}=(1,\varphi^{-1},\cdots,\varphi^{-(n-1)})\circ\mathrm{INTT}(\boldsymbol{A})$

```
1. t=n
2. m=2
3. WHILE m<=n DO
4.   t=t/2
5.   z=φ^{−m/2} //完成预计算后该语句可省略
6.   FOR j=0 TO t−1 DO
7.     FOR i=0 TO n−1 BY 2*t DO
8.       u=A[i+j]
9.       v=A[i+j+t]
10.      A[i+j]=(u+v)%M
11.      A[i+j+t]=((u−v)*z)%M
12.    z=z*φ^{−m} //可预计算 z,并按调用顺序存储在数组中
13.  m=m*2
14. FOR i=0 TO n−1 DO
15.   a[i]=(n^{−1}*A[i])%M
16. Bitreverse(a)
17. RETURN a
```

附录 B. 模整数实现算法.

为使算法描述简洁,附录 B 中$\mathbb{Z}_M=\{0,\cdots,M-1\}$. 格密码领域模整数算法的输入通常是由$\mathbb{Z}_M$中两个数的乘积,因此本文假设模整数算法的输入均满足 $x<(M-1)^2$.

算法 B1. Montgomery(a,b,M).

输入:自然数 a,b,M. $M>2$,且 $\gcd(M,2)=1$,a,$b<M$

输出:正整数 u,$u=(r^{-1}ab \bmod M)$

预计算

```
1. k=⌈log_2 M⌉
2. r=2^k
3. r^{−1}=Inverse(r,M) //r 模 M 的逆
4. M'=(r*r^{−1}−1)/M
```

约化

```
1. t=a*b
2. u=(t+(t*M' mod r)*M)/r
3. IF u<M THEN
4.   RETURN u
5. ELSE RETURN u−M
```

算法 B2. Barrett(x,M).

输入:自然数 x,M. $M>2$,且 $\gcd(M,2)=1$,$x<M^2$

输出:正整数 t,$t=(x \bmod M)$

预计算

```
1. 选择整数 k,满足 2^k>M
2. r=⌊4^k/M⌋
```

约化

```
1. t=x−⌊x*r/4^k⌋*M
2. IF t<M THEN
3.   RETURN t
4. ELSE RETURN t−M
```

算法 B3. 模 2^k+1 算法.

输入:自然数 x,正整数 k;$x<(M-1)^2$

输出:正整数 u,$u=(x \bmod 2^k+1)$

```
1. u=(x&(2^k−1))−(x≫k)
2. RETURN u
```

算法 B4. 模 2^k-1 算法.

输入:自然数 x,正整数 k;$x<(M-1)^2$

输出:正整数 u,$u=(x \bmod 2^k-1)$

```
1. u=(x&(2^k−1))+(x≫k)
2. RETURN u
```

附录 C. 模多项式实现算法.

附录 C 中假设模多项式的输入 $a(x)$的次数界为 $2N$. $f(x)$为 N 次首一多项式.

算法 C1. SM-MPA($\boldsymbol{a}$,$\boldsymbol{f}$,N).

输入:向量 $\boldsymbol{a}$,$\boldsymbol{f}$,正整数 N

输出:向量 $\boldsymbol{a}$. 满足 $a(x)=(a(x) \bmod f(x))$

```
1. FOR i=2N−1 TO N DO
2.   FOR j=N−1 TO 0 DO
3.     a[i+j−N]=a[i+j−N]−a[i]*f[j]
4. RETURN a
```

算法 C2. Sub-MPA($\boldsymbol{a}$,N).

输入:向量 $\boldsymbol{a}$,正整数 N

输出:向量 $\boldsymbol{a}$. 满足 $a(x)=(a(x) \bmod x^N+1)$

```
1. FOR i=N−1 TO 0 DO
2.   a[i]=a[i]−a[i+N]
3. RETURN a
```

附录 D.　舍入计算实现算法.

算法 D1.　Trivial-RA(x,p,q).

输入：自然数 x，正整数 p,q；$q>p,x<q$

输出：舍入值 u；$u=\lfloor(p/q)\cdot x\rceil \bmod p$

预计算

1.　$temp=p/q$　//浮点数

舍入计算

1.　$u=\lfloor x*temp+0.5\rfloor$

2.　RETURN u

算法 D2.　AS-RA(x,p,q,t).

输入：自然数 x，正整数 p,q；$q=p\times 2^t$，$x<q$

输出：舍入值 u；$u=\lfloor(p/q)\cdot x\rceil \bmod p$

1.　$u=\lfloor(x+2^{t-1})\gg t\rfloor$

2.　RETURN u

JIANG Zi-Ming, Ph. D. candidate. Her main research interest focuses on lattice-based cryptography.

ZHOU Yong-Bin, Ph. D., professor, Ph. D. supervisor. His main research interests include theories and technologies of network and information security.

ZHANG Rui, Ph. D., professor, Ph. D. supervisor. His main research interests focus on information security, cryptography, cryptographic engineering.

Background

With in-depth study and rapid development of quantum computers, traditional public-key cryptosystems based on the integer factorization problem or discrete logarithm problem would be insecure in the foreseeable future. Lattice-based cryptosystem is one of the most promising candidates for quantum-secure public key cryptography. In this work, we focus on ring learning with rounding (RLWR) and module learning with rounding (MLWR) problems in lattice-based cryptography field, which are the fundamental tools for constructing post-quantum lattice-based cryptography primitives and has been extensively applied to some basic cryptosystems.

The implementation of RLWR/MLWR consists of three basic operations: polynomial multiplication, modular reduction and rounding, among which polynomial multiplication is the most time-consuming operation. There are a variety of implementation methods of polynomial multiplication with different efficiencies which are applicable to different parameters. Number theoretic transform (NTT) is the most popular algorithm to speed up polynomial multiplication in lattice-based cryptography because of its low theoretical asymptotic complexity. However, most of the existing implementations of NTT require that the parameter q in RLWR/MLWR can be used as NTT modulus. As for the other basic operations, the existing RLWR/MLWR-based schemes mainly use the power-of-two modulus, so that some special implementation methods can be used to calculate modular reduction and rounding. Unfortunately, none of these methods can be applied to general parameters. In summary, most of the existing implementations are only applicable to some certain RLWR/MLWR-based schemes with a specific set of parameters. The efficiency of RLWR/MLWR with different parameters may vary greatly but most of the existing RLWR and MLWR schemes have not given consideration to the implementation efficiency of all basic operations. There is a lack of RLWR/MLWR parameter selection method for the purpose of achieving high efficiency.

In order to solve the above problems, we propose the general and efficient implementation algorithms of RLWR and MLWR, as well as the parameters selection methods of RLWR and MLWR to achieve better implementation efficiency. In this work, we first clarify the conditions of NTT-based polynomial multiplication algorithms. As a result, we can also use NTT to accelerate polynomial multiplication when the modulus of RLWR/MLWR does not meet the requirements of NTT modulus but meets the conditions in this work. We then present a new rounding implementation algorithm that is 11% more efficient than the traditional general rounding algorithm. Based on this, we propose efficient implementation algorithms of RLWR and MLWR for general parameters, each of which is an optimized combination of the implementation methods of polynomial multiplication, modular reduction and rounding. When it is applied to Saber scheme, the efficiency of Saber key encapsulation is improved about 52% on 64-bit Intel i7 platform without compiler optimization instruction. We finally compare the efficiency of RLWR/MLWR with different parameters and give some suggestions for parameter selection to make RLWR/MLWR more efficient.

This work is supported in part by the National Natural Science Foundation of China (Nos. 61632020, U1936209 and 62002353) and the Beijing Natural Science Foundation (No. 4192067).