基于 Cortex-M4 的 CNTR/CTRU 密钥封装高效实现

魏汉玉[1)] 郑婕妤[1)] 赵运磊[1),2)]

[1)](复旦大学计算机科学技术学院 上海 200433)

[2)](密码科学技术国家重点实验室 北京 100036)

摘 要 量子计算技术的迅猛发展对现有的公钥密码体制造成了极大的威胁,为了抵抗量子计算的攻击,后量子密码成为当前密码学界的研究热点.目前,物联网的安全问题备受关注,ARM Cortex-M4 作为低功耗嵌入式处理器,被广泛应用于物联网设备中,在其上部署后量子密码算法将为物联网设备的安全提供更加可靠的保障.CNTR 和 CTRU 是我国学者提出的 NTRU 格基密钥封装方案,相比于基于 LWE 技术路线的格基密钥封装方案在安全性和其他性能上具有综合优势,并在我国密标委得到立项.本文工作首次在 ARM Cortex-M4 平台上高效紧凑地实现了 CNTR 和 CTRU 方案,充分利用单指令多数据(Single Instruction Multiple Data,SIMD)指令,调整运算结构和指令安排,优化核心的多项式运算,从而在算法实现速度和堆栈空间上进行全面优化升级.本文主要工作如下:本文首次在 ARM Cortex-M4 上实现耗时模块多项式中心二项分布采样,采样速度提升 32.49%;使用混合基数论变换(Number Theoretic Transform,NTT)加速非 NTT 友好多项式乘法运算,充分利用浮点单元(Floating-Point Unit,FPU)寄存器,在 NTT 实现中采用层融合技术,最大化减少加载和存储等耗时指令使用,使得正向 NTT 和逆向 NTT 的速度分别提升 84.24%、81.15%;通过 NTT 过程系数范围分析进行延迟约减,进而减少约减次数,并使用改进的 Barrett 约减和 Montgomery 约减技术实现降低约减汇编指令条数;使用循环展开技术实现多项式求逆,优化多项式求逆这一耗时过程,速度优化率为 68.85%;针对解密过程中的非 NTT 友好素数模数多项式环乘法,采用多模数 NTT 和中国剩余定理(Chinese Remainder Theorem,CRT)结合的方法进行加速,完成解密过程 96.26%的速度提升;使用空间复用的方法优化堆栈空间,CNTR 和 CTRU 的堆栈空间分别减少了 29.86%、28.17%.实验结果表明:提出的优化技术大幅提升了算法实现效率,与 C 参考实现相比,CNTR 和 CTRU 的整体速度优化率分别为 85.54%、85.56%.与其他格基密钥封装方案最新 ARM Cortex-M4 实现相比,本文的优化实现在速度、空间和安全性上具有综合性的优势.

关键词 后量子密码;密钥封装方案;数论变换;多项式运算;ARM Cortex-M4 实现

中图法分类号 TP309 **DOI 号** 10.11897/SP.J.1016.2024.00589

Efficient Implementation of CNTR/CTRU Key Encapsulation Mechanism Based on Cortex-M4

WEI Han-Yu[1)] ZHENG Jie-Yu[1)] ZHAO Yun-Lei[1),2)]

[1)](*College of Computer Science and Technology, Fudan University, Shanghai* 200433)

[2)](*State Key Laboratory of Cryptology, Beijing* 100036)

Abstract The rapid advancement of quantum computing technology poses an imminent and formidable threat to the foundations of our existing public key cryptography systems. As researchers grapple with the urgent need to safeguard our digital infrastructure against the imminent threat of quantum attacks, the concept of Post-Quantum Cryptography (PQC) has emerged as a dynamic and thriving field of study. By exploring innovative cryptographic techniques and mathematical constructs,

收稿日期:2023-05-22;在线发布日期:2023-12-29.本课题得到国家重点研发计划基金(2022YFB2701600)、密码科学技术国家重点实验室面上课题基金(MMKFKT202227)、上海市科委技术标准基金(21DZ2200500)、上海市协同创新基金(XTCX-KJ-2023-54)、上海市科委区块链关键技术攻关专项基金(23511100300)资助.**魏汉玉**,硕士研究生,主要研究方向为后量子密码、密码工程.E-mail:hywei22@m.fudan.edu.cn.**郑婕妤**,博士研究生,主要研究方向为后量子密码、密码工程.**赵运磊**(通信作者),博士,特聘教授,主要研究领域为后量子密码、密码协议、密码工程、计算理论.E-mail:ylzhao@fudan.edu.cn.

researchers in the field of PQC strive to ensure the long-term security and resilience of our digital communication systems in the face of the impending quantum revolution. At present, the security issues of the Internet of Things (IoT) have attracted much attention. ARM Cortex-M4, as a low-power embedded processor, is widely used in IoT devices, thus deploying post-quantum cryptography algorithms on it will provide more reliable guarantees for the security of IoT devices. CNTR and CTRU are NTRU lattice-based Key Encapsulation Mechanisms (KEM) proposed by Chinese scholars. These schemes offer comprehensive advantages in terms of security and performance compared to lattice-based KEMs based on the Learning With Errors (LWE) technical route. They have received funding from the Cryptography Standardization Technical Committee (CSTC) in China. The primary focus of this research paper is to efficaciously and succinctly implement the CNTR and CTRU schemes on the ARM Cortex-M4 platform. Leveraging the power of Single Instruction Multiple Data (SIMD) instructions, strategically adjusting the operation structure, and optimizing the instruction arrangement, we have succeeded in significantly augmenting the speed and optimizing the stack usage of these algorithms. The main contributions of this paper include: Firstly, this paper introduces the implementation of the time-consuming module polynomial Central Binomial Distribution (CBD) sampling on ARM Cortex-M4 for the first time. This implementation accelerates the sampling process by an impressive 32.49%. Additionally, mixed-radix Number Theoretic Transform (NTT) is employed to expedite NTT-unfriendly polynomial multiplication. By fully utilizing the Floating-Point Unit (FPU) registers and employing layer merging in NTT to minimize time-consuming operations like load and store, the speed of NTT and inverse NTT experiences substantial improvements of 84.24% and 81.15% respectively. Moreover, this research employs coefficient range analysis during the NTT process to delay reduction, thereby reducing the number of reductions required. Furthermore, it utilizes improved Barrett reduction and Montgomery reduction techniques to minimize the number of assembly instructions involved. Polynomial inversion is achieved through loop unrolling, optimizing the time-consuming process and resulting in a speed optimization rate of 68.85%. To accelerate the decryption process, which involves NTT-unfriendly prime modulus polynomial ring multiplication, a combination of multi-moduli NTT and the Chinese Remainder Theorem (CRT) is employed, providing a remarkable speed improvement of 96.26%. Additionally, space multiplexing is utilized to optimize stack usage, resulting in stack usage reductions of 29.86% and 28.17% for CNTR and CTRU respectively. Experimental results demonstrate that the proposed optimization techniques significantly enhance algorithm efficiency. Compared to the C reference implementation, CNTR and CTRU exhibit overall speed optimization rates of 85.54% and 85.56% respectively. Furthermore, in a comparative analysis with the state-of-the-art implementation of other lattice-based key encapsulation mechanisms on the ARM Cortex-M4 platform, the optimized implementation presented in this paper clearly demonstrates comprehensive advantages across multiple dimensions, including speed, efficient utilization of space, and robust security measures.

Keywords post-quantum cryptography; key encapsulation mechanism; number theoretic transform; polynomial arithmetic; ARM Cortex-M4 implementation

1 引 言

近年来,量子计算技术突飞猛进.与传统计算机相比,量子计算机针对特定问题,特别是当前公钥密码所基于的大数分解和离散对数等问题,能够实现算力呈指数级规模拓展和爆发式增长,利用量子叠加和纠缠特性带来的强大并行处理来超越经典计算

机的性能.

传统公钥密码体制主要基于离散对数问题和大整数分解问题.虽然这些困难问题目前不能在传统计算机中被攻破,但是在量子计算机中,Shor 算法[1]可以在多项式时间内破解传统公钥密码体制的困难问题.因此,研发能够抵抗量子攻击的新型公钥密码算法——后量子密码算法,得到了学术界和工业界的广泛关注.

为应对量子计算机对传统公钥密码体制的威胁,2016 年美国国家标准与技术研究院(National Institute of Standards and Technology,NIST)向全世界学者征集后量子密码算法标准,包括公钥加密(Public Key Encryption,PKE)、密钥封装方案(Key Encapsulation Mechanism,KEM)和数字签名,经过三轮竞赛评选,得到7个最终算法和8个候选算法[2].2022 年,NIST 公布了 4 个拟标准化的算法[3],其中 3 个是基于格的.基于格的密码算法设计主要基于一般格、理想格、模格和 NTRU 格.格的困难问题通常分为以下两类.第一类是{R,M}LWE/LWR 困难问题[4-8],包括容错学习(Learning with Errors,LWE)问题和舍入学习(Learning with Rounding,LWR)问题,其中 R 表示环(Ring)结构,M 表示模(Module)结构;第二类是 NTRU 格相关问题[9-11].

在 NIST 征集的后量子密码算法中,基于 NTRU 格的方案具有重要的地位.具体而言,NIST 拟标准化的数字签名方案之一 Falcon[12]基于 NTRU 格构造.此外,在 NIST 后量子密码算法征集第三轮中,NTRU KEM(包括 NTRU-HRSS 和 NTRUEncrypt)[13]是 4 个决赛密钥封装方案之一,NTRU Prime[14]是 5 个候选方案之一.尽管目前 NIST 选择 Kyber[15]作为拟标准化密钥封装方案,而 Kyber 受到复杂专利因素的影响.自 NTRU 格提出至今,对 NTRU 格基密钥封装方案的攻击对其安全性造成的影响十分有限.并且,基于 NTRU 格构造的方案结构简单,便于实现,加解密算法的运算速度快,例如文献[13-14]的方案.基于 NTRU 格的方案密钥长度灵活,而基于 MLWE 的密钥封装方案,例如 Kyber,密钥长度是固定的.同时,NTRU 的相关核心专利已失效,利于在实际应用中进行部署应用.因此,基于 NTRU 的密钥封装反而在实际中得到加速应用.

早在 2008 年,IEEE Std 1363.1 标准便开始采用包括 NTRUEncrypt 在内的格密码算法作为标准算法.在 2011 年,NTRUEncrypt 方案便已入选 X9.98 标准并应用于金融服务之中.在 2015 年,欧盟的 PQCRYPTO 项目(Horizon 2020 ICT-645622)考虑采纳 NTRU 相关算法作为欧洲的标准之一.在 2019 年,谷歌开展 CECPQ2 实验.不同的是,这次谷歌将 NTRU-HRSS 算法嵌入到 TLS1.3 之中,并测试其性能,由于免受专利威胁,谷歌 CECPQ2 实验进展顺利.在 2021 年 5 月,OpenBSD 便开始增加了 Streamlined NTRU Prime-761 算法结合椭圆曲线经典算法 ECC,应用在 IPsec 上.自 2022 年 4 月起,国际标准 OpenSSH 更新了版本 9.0 之后,OpenSSH 便以默认模式采用了 NTRU Prime 方案结合 X25519 ECDH 方案的混合模式,以抵抗"先截获后解密"攻击导致的前向安全风险.

越来越多的研究者聚焦于 NTRU 格基方案设计和实现,已知的基于 NTRU 格的密钥封装方案存在以下问题:(1)秘密范围小:传统基于 NTRU 格的方案秘密取值为{0,−1,1},限制了方案的安全级别[13-14,16];(2)带宽大:模数 q 较大,且不能压缩密文,不适用于物联网等资源受限设备[9,13,17-18];(3)密钥生成慢:密钥生成的多项式求逆 $h=g/f$ 计算耗时多[9,13-14,18].基于 NTRU 格的密钥封装方案[9,13-14,16-18]普遍存在上述问题之一.我国学者提出的基于 NTRU 格的密钥封装方案 CNTR 和 CTRU[19]在安全性、通信带宽、错误率和实现效率方面表现得性能均衡,其允许更大的密钥范围,实现了尺度化密文压缩,带宽小实现效率高,适用于嵌入式平台实现,并且具有选择密文攻击(Chosen Ciphertext Attack,CCA)安全性.尽管 CNTR 和 CTRU 并非 NIST 拟标准化方案,但目前已在我国密码行业标准化技术委员会(以下简称"密标委")立项[20],对我国丰富抗量子格基密码技术路线具有实际意义.

NTRU 格基密钥封装方案特别适用于微处理器,因为密钥长度较小,计算性能特别快,封装和解封装在几毫秒完成.而小型嵌入式设备一般内存小、功耗低.ARM Cortex-M4 是 NIST 推荐优化实现的微处理器平台,优点是有足够的空间支持公钥算法,同时,相对庞大的计算而言,空间小且便宜.ARM Cortex-M4 是一种低功耗、高性能的嵌入式处理器,常用于物联网设备、传感器网络、工业和家居自动化、医疗保健和健康应用等场景.物联网是数字经济的重要组成部分,是国家科技创新的重要方向.随着物联网技术的蓬勃发展,安全问题越来越凸显,而密码算法是物联网安全的基础,密码算法的安全性直接影响物联网设备和通信的安全性.在嵌入式设备上部署安全高效的轻量级密码算法为物联网安

全带来重要的意义，提高安全性和隐私性，抵御量子计算攻击的威胁.因此，后量子格基密码算法的 ARM Cortex-M4 高效实现和测试对我国推进后量子密码算法的标准化和保障国家信息安全具有重要的意义.

为了应对量子计算的挑战，未来国家后量子密码算法标准的制定和实际部署需要我国科研机构和产业的共同努力，而在嵌入式设备部署高效安全的后量子密码算法是重要的研究目标.本文针对 CNTR 和 CTRU 方案完成嵌入式平台 ARM Cortex-M4 的优化实现，充分利用 Cortex-M4 的数字信号处理(Digital Signal Processing，DSP)指令集，实现单指令多数据(Single Instruction Multiple Data，SIMD)的并行效果，对算法底层的多项式运算以及所需堆栈空间进行优化.具体而言，本文的主要贡献有以下几点：

(1)多项式采样是方案较为耗时操作之一，本文首次在 ARM Cortex-M4 上实现多项式中心二项分布采样，调整运算结构实现四倍并行，将多项式采样的速度提升 32.49%；

(2) CNTR 和 CTRU 使用混合基数论变换(Number Theoretic Transform，NTT)加速多项式乘法运算，采用层融合技术，正向 NTT 使用 4+3+1 层融合完成 8 层变换.实现时使用浮点单元(Floating-Point Unit，FPU)寄存器暂存旋转因子和中间值，以融合更多的 NTT 层，从而减少加载旋转因子和多项式系数的内存访问时间，将多项式乘法的速度提升 83.65%；

(3)使用六条指令的二并行有符号 Barrett 约减 Cortex-M4 实现，相较于之前的实现减少了两条指令.使用改进的有符号 Montgomery 约减.通过分析正向 NTT 和逆向 NTT 过程中系数范围进行延迟约减，减少非必要的模约减以提升效率；

(4)针对解密过程中的非 NTT 友好素数模数多项式环乘法，使用多模数 NTT 与 CRT 结合的方法计算多项式乘法.选取 NTT 友好素数 $q=3457$，$q'=7681$.首先这两个模数满足 NTT 和 CRT 结合计算 $q_2=1024$ 多项式乘法的要求，其次选取这两个模数可以复用加密过程的 $q=3457$ 的 NTT 代码，并且 $q'=7681$ 也是我国抗量子密码算法竞赛获奖算法 Aigis-KEM[21] 以及 NTTRU[16] 所采用的模数.解密过程的速度优化了 96.26%；

(5)通过重复利用已经申请的多项式变量来完成不同的运算操作，这种空间复用的方法减少需要申请的多项式变量数量，从而减少所占用的堆栈空间，相比于 CNTR 和 CTRU 的 C 参考实现，我们的 Cortex-M4 实现的堆栈空间分别减少 29.86%和 28.17%.

2 相关工作

NIST 后量子密码标准化进程中，基于 NTRU 格的密钥封装方案广受关注，并且衍生了多个变体优化方案[9,14,16-18].Liang 等人[19]从理论上分析了 NTRU 格基密钥封装方案的优势和存在的挑战，提出了新的 NTRU 格基密钥封装方案 CNTR 和 CTRU，这是紧凑高效的密钥封装方案，使用创新的加密算法，打破了 NTRU 格基密钥封装方案密文压缩的限制，在单一多项式的情况下压缩密文.

目前，后量子密码高效实现已经成为密码学领域的研究热点，其需要在不同平台上计算：从高性能服务器到资源受限的嵌入式设备.由于物联网设备的大规模增长[22]，在这类嵌入式设备部署密码算法显得尤为重要.并且，后量子密码算法特别适用于嵌入式设备，NIST 已将 ARM Cortex-M4 指定为后量子密码算法的主要微处理器优化平台.

国内外学者们提出了很多针对后量子密码算法的 ARM Cortex-M4 优化实现，使用 pqm4[23] 提供的基准测试框架，评估算法运行的时钟周期和堆栈空间.后量子密码主流算法多数基于格，其困难问题通常分为两类.第一类是{R，M}LWE/LWR 问题[4-8]，代表算法有 Kyber[15]、Saber[24]；第二类是 NTRU 格相关问题[9-11]，代表算法有 NTRU[13]、NTRU Prime[14]、NTTRU[16].

Botros 等人[25]完成 Kyber 算法的 ARM Cortex-M4 实现，使用层融合、预计算旋转因子、打包系数等方法加速 NTT，同时优化堆栈空间，达到时间和空间的平衡.Alkim 等人[26]针对 Kyber 算法的模约减操作，将文献[25]中使用的 Montgomery 约减算法[27]优化到 2 个时钟周期，并延迟约减，减少约减次数以加速实现.Abdulrahman 等人[28]优化 Kyber 算法的 Cortex-M4 实现，采用浮点单元代替通用寄存器暂存 NTT 运算中的数据，最大化利用微处理器的空间，以便融合更多的 NTT 层，减少内存访问的时间，其中，利用 smlaw{b，t}指令优化 Barrett 约减算法[29]，比文献[24-25]使用的 Barrett 算法少 2 条指令，进一步提升约减效率.

Chung 等人[30]对于 Saber 算法中的多项式乘法，给出了 Toom-Cook[31] 和多模数 NTT 两种实现

方案，并比较了两者在时间上的性能，实验结果表明多模数 NTT 速度优势明显. Abdulrahman 等人[32]在 ARM Cortex-M4 平台完成 Saber 算法的多种实现，其中，相较于文献[30]的非屏蔽实现优化了堆栈空间，提供了抗侧信道攻击的屏蔽实现. 实验表明，在屏蔽和非屏蔽实现中多模数 NTT 速度优于 Toom-Cook.

Chung等人[30]完成了NTRU算法的ARM Cortex-M4 实现，其中使用多模数 NTT 加速多项式乘法比 Toom-Cook 更快，但并没有达到预期的效果. Paksoy 等人[33]进一步优化 NTRU 算法的实现，采用 TMVP 方法[34]加速多项式乘法，完成文献[30]中 NTRU 算法的密钥生成、加解密、密钥封装和解封装的全面加速，并且栈空间占用更少. Alkim 等人[35]针对 NTRU Prime 算法中的非 NTT 友好环乘法，比较 Good[36]和混合基 NTT 两种技巧在时间上的性能，实验结果表明混合基 NTT 速度更快. Huang 等人[37]给出首个 NTTRU 算法的 Cortex-M4 实现，引入 Plantard 约减算法[38]，利用 smulw{b,t}指令完成 16×32 位乘积约减，扩大约减的输入范围且缩小输出范围.

3　预备知识

3.1　ARM Cortex-M4

ARM Cortex-M4 作为NIST后量子密码项目推荐的评估和测试平台，是本文软件实现的目标平台. ARM Cortex-M4 是 ARMv7E-M 架构的数字信号控制器，可以满足高性能通用代码处理以及数字信号处理应用的需求，共有 14 个可用的 32 位通用目的寄存器. 另外，Cortex-M4 支持浮点计算，有 32 个 32 位的 FPU 寄存器，用于暂存频繁使用的数据，避免过多的内存加载和存储操作. 对于 Cortex-M4 而言，内存加载(ldr 指令)和存储(str 指令)是极其耗时的，一条 ldr 或 str 指令占用 1 到 2 个时钟周期，而用于 FPU 寄存器的数据传输指令 vmov 仅需要 1 个时钟周期. 同时，使用 FPU 可以解决 Cortex-M4 中通用寄存器紧缺的情况，可以有更多的空间来暂存中间数据.

ARM Cortex-M4核心指令集包含基本的Thumb-1、Thumb-2 指令，以及数字信号处理扩展指令集. DSP 指令集可以实现同时对 4 个 8 位或 2 个 16 位整数进行操作. ARM Cortex-M4 可以实现 SIMD 的并行效果，通过优化代码结构和指令安排，实现算法性能的提升. 例如，smul{b,t}{b,t}指令将指定的半字相乘；smla{b,t}{b,t}指令将半字相乘的结果再累加到指定的寄存器中；smuad{,x}指令实现两个半字相乘且乘积相加，该指令适合用于一次多项式的点乘；sxth16 指令将两个字节有符号展开为半字. Cortex-M4 上执行一条指令一般需要一个时钟周期，而内存加载或存储操作有时会再多增加一个时钟周期的消耗，故需要合理安排指令以达到效率最优.

3.2　CNTR 算法和 CTRU 算法介绍

基于NTRU格的密钥封装方案CNTR和CTRU在安全性、带宽、错误率和计算效率等方面有较好的性能表现，甚至优于基于{R,M}LWE 的密钥封装方案，如 Kyber. CNTR 和 CTRU 允许更大的密钥范围，实现了尺度化密文压缩，具有 CCA 安全性，并且实现速度优于 NTRU. CNTR 和 CTRU 的错误率足够低，在推荐参数下比 Kyber 的错误率更低.

CNTR和CTRU的核心组成部分是选择明文攻击下不可区分性(Indistinguishability under Chosen Plaintext Attack，IND-CPA)安全的公钥加密方案 CNTR.PKE 和 CTRU.PKE，以及选择密文攻击下不可区分性(Indistinguishability under Chosen Ciphertext Attack，IND-CCA)安全的密钥封装方案，记作 CNTR. KEM 和 CTRU. KEM. 密钥封装方案是通过 FO 转换(Fujisaki-Okamoto Transformation)的变体和前缀哈希($\mathrm{FO}^{\not\perp}_{ID(pk),m}$)构造出来的[19]. CNTR 的安全性基于 NTRU 假设和 RLWR 假设；CTRU 的安全性基于 NTRU 假设和 RLWE 假设.

算法 1.　IND-CPA安全的CNTR.PKE.

密钥生成 KeyGen()

1. $f',g \leftarrow \Psi_1$
2. $f := pf'+1$
3. IF f is not invertible in $\mathcal{R}_q$, RESTART.
4. $h := g/f$
5. RETURN $(pk := h, sk := f)$

加密 Enc($pk=h, m\in\mathcal{M}$)

1. $r \leftarrow \Psi_2$
2. $\sigma := hr$
3. $c := \left\lfloor \frac{q_2}{q}(\sigma + PolyEncode(m)) \right\rceil \bmod q_2$
4. RETURN c

解密 Dec($sk=f, c$)

1. $m := PolyDecode(cf \bmod \pm q_2)$
2. RETURN m

如算法 1 所示,CNTR 使用 $h=g/f$ 作为公钥,将 f 作为私钥.加密算法是将每 4 位消息编码成一个尺度化 E_8 格点,基于 RLWR 问题隐藏明文从而生成密文.该算法将消息编码为密文的最高有效位,使得压缩密文的过程并不会破坏消息的有用信息.CNTR 的解密算法是将私钥多项式和密文多项式相乘,再将多项式乘法结果传入到多项式解码算法中恢复出明文.关键点在于,不同于现有的 NTRU 格基密钥封装方案,CNTR 的模 p 消息空间在公钥 h 和密文 c 中是不存在的,因为在 CNTR 构造中不需要使用公钥和密文的模 p 消息空间来恢复消息.唯一为 p 保留位置的是密钥 f,保留形式是 $f=pf'+1$.加密和解密时分别通过算法 PolyEncode 和 PolyDecode 实现,两个算法的实现原理见文献[19].最后,使用 FO 转换的变体将 IND-CPA 安全的 CNTR.PKE 转换为 IND-CCA 安全的 CNTR.KEM,如算法 2 所示.

算法 2. IND-CCA 安全的 CNTR.KEM.

密钥生成 $KeyGen()$

1. $(pk,sk)\leftarrow$CNTR.PKE.KeyGen()
2. $z\xleftarrow{\$}\{0,1\}^l$
3. RETURN $(pk':=pk, sk':=(sk,z))$

封装 Encaps(pk)

1. $m\xleftarrow{\$}\mathcal{M}$
2. $(K,coin):=\mathcal{H}(ID(pk),m)$
3. $c:=$CNTR.PKE.Enc($pk,m;coin$)
4. RETURN (c,K)

解封装 Decaps($(sk,z),c$)

1. $m':=$CNTR.PKE.Dec(sk,c)
2. $(K',coin'):=\mathcal{H}(ID(pk),m')$
3. $\widetilde{K}:=\mathcal{H}_1(ID(pk),m')$
4. IF $m'\neq\perp$ and $c=$CNTR.PKE.Enc($pk,m';coin'$) THEN
5. RETURN K'
6. ELSE
7. RETURN $\widetilde{K}$
8. END IF

CNTR 和 CTRU 方案中所有哈希函数均使用 SHA-3 国际标准进行实例化.为了生成秘密多项式,即 f',g,r,e,需要使用 SHAKE-128 将秘密种子扩展到采样随机性.哈希函数 $\mathcal{H}$ 用 SHA3-512 实例化,将公钥的短前缀 ID(pk)和消息 m 哈希为 64 字节,其中前 32 字节用于生成共享密钥,后 32 字节作为加密算法的秘密种子.我国标准 SM3 杂凑函数的输出长度为 32 字节,如果替换为 SM3,则不能满足 CNTR/CTRU 算法的安全要求,算法安全性会受到影响.并且,如同 NIST 后量子密码标准中均采用了海绵结构的 SHAKE 扩展型哈希函数(其输出长度是任意设定的)来产生随机数和拒绝采样,而我国杂凑函数没有与 SHAKE 对应的输出长度任意设定的海绵结构杂凑标准化算法,因此也难以应用我国杂凑函数标准.故本文使用SHA-3作为底层哈希函数.

CNTR和CTRU展示了NTRU格基密钥封装的新的构造方法.其构造在NTT友好环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$.选择 $n\in\{512,768,1024\}$分别对应 NIST 推荐的安全级别 I,III 和 V,这三种维度都是同一个模数 $q=3457$(接近 Kyber 的模数 $q=3329$),便于简洁高效和高兼容实现.推荐使用 $n=768$ 的参数集,因为这一维度可以最好地达到后量子安全和实际应用性能的平衡.

CTRU 方案是 CNTR 的演化版本:增加了噪声多项式 e(算法 1 加密部分第 2 行变为 $\sigma:=hr+e$),对尺度化 E_8 格编码算法的输出做四舍五入.目前而言,CNTR 和 CTRU 是第一个将高维度 NTRU 格和低维度格编码相结合的 NTRU 格基密钥封装方案,也是第一个通过单个多项式进行尺度化密文压缩的 NTRU 格基密钥封装方案.由于 CNTR 和 CTRU 的区别对 ARM 汇编指令实现没有影响,本文重点分析 $n=768$ 的参数下 CNTR 的 ARM Cortex-M4 优化实现.

3.3 NTT 算法介绍

NTT 是快速傅里叶变换(Fast Fourier Transform,FFT)在有限域上的特殊情况,主要思想是将多项式系数转换到 NTT 域内来执行更快的多项式乘法运算.给定两个多项式 a,b,它们的乘法记作 $c=a\cdot b=\mathrm{NTT}^{-1}(\mathrm{NTT}(a)\circ\mathrm{NTT}(b))$.其中,$\mathrm{NTT}^{-1}$ 表示逆向 NTT,"$\circ$"表示 NTT 域内的点乘.使用中国剩余定理(Chinese Remainder Theorem,CRT)的思想:令 $\mathcal{R}$ 是环,如果 f 和 g 互素,那么存在环同态 ϕ:$\mathcal{R}[x]/(f(x)g(x))\cong\mathcal{R}[x]/f(x)\times\mathcal{R}[x]/g(x)$,$\phi(h)=(h \bmod f, h \bmod g)$.当 $f(x)=x^n-c$ 和 $g(x)=x^n+c$ 时,ϕ 自然变成

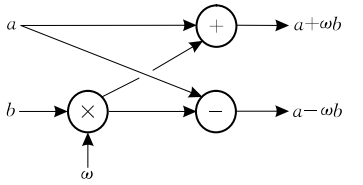
$$\phi\left(\sum_{i=0}^{2n-2}h_ix^i\right)=\left(\sum_{i=0}^{n-1}(h_i+ch_{n+i})x^i,\sum_{i=0}^{n-1}(h_i-ch_{n+i})x^i\right),$$

$$\phi^{-1}\left(\left(\sum_{i=0}^{n-1}h'_ix^i\right),\left(\sum_{i=0}^{n-1}h''_ix^i\right)\right)$$

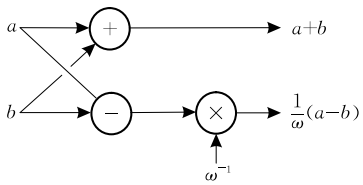
$$=\sum_{i=0}^{n-1}\frac{1}{2}(h'_i+h''_i)x^i+\sum_{i=0}^{n-1}\frac{1}{2}\frac{1}{c}(h'_i-h''_i)x^{n+i}.$$

快速傅里叶变换可以在准线性时间 $O(n\log n)$

计算 NTT. FFT-trick 乘法技巧可以表示为 $h_1h_2 \pmod{(x^{2n}-c^2)} \equiv \frac{x^n+c}{2c}(h_1 \bmod (x^n-c))(h_2 \bmod (x^n-c)) + \frac{-x^n+c}{2c}(h_1 \bmod (x^n+c))(h_2 \bmod (x^n+c))$. FFT-trick 算法通常使用 CT(Cooley-Tukey)蝴蝶变换[39]实现正向 NTT,使用 GS(Gentleman-Sande)蝴蝶变换[40]实现逆向 NTT,图 1(a)和(b)分别表示这两种变换.



(a) CT蝴蝶变换



(b) GS蝴蝶变换

图 1　快速傅里叶变换的“蝴蝶”操作

FFT-trick 算法有一个特殊的性质,即输出是比特反转(bit-reversed,brv)顺序. 对长度为 $2k$ 的数组进行比特反转,即将每个元素的索引转化成长度为 k 的二进制字符串,再反转比特串. 对于索引 i,i_j 表示第 j 个比特,比特反转 brv 表示为

$$i=\sum_{j=0}^{k-1} i_j\,2^j,\ brv_k(i)=\sum_{j=0}^{k-1} i_{k-j}\,2^j.$$

由于比特反转在软件实现时是一项代价高昂的操作,因此会尽量避免显式地执行它. 一般以比特反转顺序在 NTT 域中保留值. 再执行逆向 NTT 恢复正常顺序,又因为 NTT 中的乘法和加法是逐元素运算,故比特反转对多项式乘法的顺序没有影响.

3.4　CNTR/CTRU 的混合基 NTT

上述介绍的 FFT-trick 算法都作用在多项式环 $\mathbb{Z}[x]/(x^{2^k}\pm 1)$ 上,每个 NTT 层分解为两部分,称为基 2(Radix-2)NTT. 然而,通常还需要使用不同的基来做 FFT-trick 变换,例如基 3(Radix-3)NTT. 基 3NTT 将环 $\mathbb{Z}[x]/(x^{3^{k+1}}-c^3)$ 分解为 $\mathbb{Z}[x]/(x^{3^{k+1}}-c^3) \cong \mathbb{Z}[x]/(x^{3^k}-c) \times \mathbb{Z}[x]/(x^{3^k}-\omega_3 c) \times \mathbb{Z}[x]/(x^{3^k}-\omega_3^2 c)$.

基 3NTT 使用 CT 蝴蝶变换的形式分解 $\mathbb{Z}[x]/(x^{3^{k+1}}\pm c^3)$ 记作下式,其中 $0\le i<3^k$:

$$\hat{a}=a_i+ca_{i+3^k}+c^2a_{i+2\cdot 3^k},$$

$$\hat{a}_{i+3^k}=a_i+\omega_3 ca_{i+3^k}+\omega_3^2c^2a_{i+2\cdot 3^k},$$

$$\hat{a}_{i+2\cdot 3^k}=a_i+\omega_3^2 ca_{i+3^k}+\omega_3 c^2a_{i+2\cdot 3^k}.$$

逆向基 3NTT 使用 GS 蝴蝶变换记为

$$a_i=3^{-1}(\hat{a}_i+\hat{a}_{i+3^k}+\hat{a}_{i+2\cdot 3^k}),$$

$$a_{i+3^k}=3^{-1}c^{-1}(\hat{a}_i+\omega_3^2\hat{a}_{i+3^k}+\omega_3\hat{a}_{i+2\cdot 3^k}),$$

$$a_{i+2\cdot 3^k}=3^{-1}c^{-2}(\hat{a}_i+\omega_3\hat{a}_{i+3^k}+\omega_3^2\hat{a}_{i+2\cdot 3^k}).$$

对于 CNTR 的环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$,我们采用基 2NTT 结合基 3NTT 进行变换,即混合基 NTT. 首先,进行一层分解,将原本的环分解为两个可以做基 2NTT 的环 $\mathbb{Z}_q[x]/(x^{n/2}-\zeta_1)$ 和 $\mathbb{Z}_q[x]/(x^{n/2}-\zeta_2)$,多项式 $f\in\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 的分解方式如下:

$$\begin{cases} f \bmod x^{n/2}-\zeta_1 = (f_0+\zeta_1 f_{n/2})+\cdots+(f_{n/2-1}+\zeta_1 f_{n-1})x^{n/2-1}, \\ f \bmod x^{n/2}-\zeta_2 = (f_0+f_{n/2}-\zeta_1 f_{n/2})+\cdots+(f_{n/2-1}+f_{n-1}-\zeta_1 f_{n-1})x^{n/2-1}. \end{cases}$$

随后,对 $\mathbb{Z}_q[x]/(x^{n/2}-\zeta_1)$ 和 $\mathbb{Z}_q[x]/(x^{n/2}-\zeta_2)$ 利用 CRT 进行基 2NTT. 对于每个基 2NTT 层,CRT 同构形如:$\mathbb{Z}_q[x]/(x^{2m}-r^2)\cong\mathbb{Z}_q[x]/(x^m-r)\times\mathbb{Z}_q[x]/(x^m+r)$,其中 r 为 ζ 的某次幂,对于 $f'\in\mathbb{Z}_q[x]/(x^{2m}-r^2)$,可得:

$$\begin{cases} f_l = f' \bmod x^m-r \\ \quad =(f'_0+rf'_m)+\cdots+(f'_{m-1}+rf'_{2m-1})x^{m-1}, \\ f_r = f' \bmod x^m+r \\ \quad =(f'_0-rf'_m)+\cdots+(f'_{m-1}-rf'_{2m-1})x^{m-1}. \end{cases}$$

对于 $n=768$,进行 6 层基 2NTT,直至得到 $n/6$ 个模 6 次多项式的项,如 $\mathbb{Z}_q[x]/(x^6-\zeta^3)$,此时,再做一层基 3NTT,以 $\mathbb{Z}_q[x]/(x^6-\zeta^3)$ 为例,有以下 CRT 同构:$\mathbb{Z}_q[x]/(x^6-\zeta^3)\cong\mathbb{Z}_q[x]/(x^2-\zeta)\times\mathbb{Z}_q[x]/(x^2-\rho\zeta)\times\mathbb{Z}_q[x]/(x^2-\rho^2\zeta)$,其中 $\rho=\zeta^{n/2} \bmod q$. $\mathbb{Z}_q[x]/(x^6-\zeta^3)$ 中的多项式 f'' 可表示为:$f''=f_a+f_bx^2+f_cx^4$,其中 f_a,f_b,f_c 为线性多项式. 由于 $\rho^3=1 \bmod q$,可得

$$\begin{cases} f_x = f'' \bmod x^2-\zeta = f_a+\zeta f_b+\zeta^2 f_c, \\ f_y = f'' \bmod x^2-\rho\zeta = f_a+\rho\zeta f_b+(\rho\zeta)^2 f_c, \\ f_z = f'' \bmod x^2-\rho^2\zeta = f_a+\rho^2\zeta f_b+(\rho^2\zeta)^2 f_c. \end{cases}$$

基 3NTT 做完后,得到 $n/2$ 个模 2 次多项式的项. 至此,CNTR 的环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上正向 NTT 完成,总计 1 层分解、6 层基 2NTT 和 1 层基 3NTT. 逆向 NTT 是正向变换的逆过程,因此逆向 NTT 同样为 6 层基 2NTT 和 1 层基 3NTT.

3.5　多项式采样

CNTR 中多项式采样使用中心二项分布(Cen-

tered Binomial Distribution,CBD)B_η,其中 $\eta=2$ 或 $\eta=3$. 定义 B_η:采样$(a_1,\cdots,a_\eta,b_1,\cdots,b_\eta)\leftarrow\{0,1\}^{2\eta}$ 输出$\sum_{i=1}^{\eta}(a_i-b_i)$.

针对 CNTR 方案具体而言,算法 3 的 CBD 函数定义了根据 B_η 确定性从伪随机函数生成的 192η 字节,采样多项式 $f\in\mathcal{R}_q$ 的方法. 此处描述中,固定 $n=768$.

算法 3. CBD_η: $\mathcal{B}^{192\eta}\rightarrow\mathcal{R}_q$.

输入:字节流 $B=(b_0,b_1,\cdots,b_{192\eta-1})\in\mathcal{B}^{192\eta}$

输出:多项式 $f\in\mathcal{R}_q$

1. $(\beta_0,\cdots,\beta_{1536\eta-1}):=BytesToBits(B)$
2. FOR $i=0,\cdots,767$ DO
3. $\quad a:=\sum_{j=0}^{\eta-1}\beta_{2i\eta+j}$
4. $\quad b:=\sum_{j=0}^{\eta-1}\beta_{2i\eta+\eta+j}$
5. $\quad f_i:=a-b$
6. END FOR
7. RETURN $f_0+f_1X+f_2X^2+\cdots+f_{767}X^{767}$

本文主要针对 $\eta=2$ 的情况做优化实现,故以下描述均以 $\eta=2$ 为例. 算法 3 的第 2~6 行,从比特串低位依次取 2 比特位并将这 2 比特值相加,所得的结果是这 2 比特位的汉明重量,即这 2 比特位中 1 的个数,其范围是{0,1,2}. 每个循环取两次 2 位,将其汉明重量分别记录为 a 和 b, a 和 b 做有符号减法的结果作为多项式系数 f_i.

在实际实现中,多项式采样分为两部分,第一部分是加载(load)函数,用来拼接字节流,每 4 字节以小端存储的方式转换成 32 位无符号数,对于输入的 192η 字节,共转化为 96 个 32 位无符号数;第二部分是 CBD 函数,对 load 函数返回的 96 个 32 位无符号数,逐个处理,每个 32 位数可以生成 8 个多项式系数. 如算法 4 所示,第 2、3 行计算比特串每 2 比特位的汉明重量,结果存放在变量 d 中. 因为 0x55555555 的二进制形式是 01010101010101010101010101010101,故 d 取 t_i&0x55555555 就是取 t_i 的奇数位,而 t_i 右移 1 位再 &0x55555555 就是取 t_i 的偶数位. 两者相加就是 t_i 每 2 位的汉明重量. 对记录汉明重量的 d 每次取 4 位,a 取低 2 位,b 取高 2 位,a 和 b 相减的差作为多项式系数 f_i. 实现中需要注意数据长度、符号位等对运算的影响,t_i 和 d 是 32 位无符号数,a、b 和 f_i 都是 16 位有符号数. 给 a、b 赋值时采用按位与(&0x3),故取 a,b 的最低 2 位,其余位均为 0,即符号位为 0. 实际上,算法 4 是将算法 3 的思想用 C 编程的技巧进行实现,理解底层比特位的存储形式对汇编实现是极其重要的.

算法 4. C 编程实现 CBD_2.

输入:32 位无符号数组 $t[96]=\{t_0,t_1,\cdots,t_{95}\}$

输出:多项式 $f=(f_0,f_1,f_2,\cdots,f_{767})$

```
1.  FOR i=0,…,95 DO
2.      d=t_i&0x55555555
3.      d+=(t_i≫1)&0x55555555
4.      FOR j=0,…,7 DO
5.          a=(d≫(4*j+0))&0x3
6.          b=(d≫(4*j+2))&0x3
7.          f_{8i+j}=a-b
8.      END FOR
9.  END FOR
10. RETURN f=(f_0,f_1,…,f_767)
```

4 CNTR/CTRU 方案 Cortex-M4 实现

CNTR 和 CTRU 算法的 ARM Cortex-M4 实现基于 C 参考实现[19]和 pqm4[23]基准测试框架,并且对重要的多项式运算模块(例如,多项式采样、正向 NTT、逆向 NTT、多项式求逆等)使用 Thumb 和 DSP 汇编指令集进行优化,使用 C 和 ARM 交叉编译,从而完成CNTR和CTRU方案的整体实现. 在本章节中,主要阐述多项式运算模块的 ARM Cortex-M4 实现技巧,通过仔细优化运算结构,充分利用 SIMD 指令并行特性,提升算法性能.

4.1 多项式采样

本文充分利用 ARM Cortex-M4 汇编底层小端存储的优势. 在多项式采样的第一部分,C 实现平台需要使用 load 函数将输入的字节流每 4 个字节为一组,以小端存储的方式转换成 32 位无符号数;而在 ARM Cortex-M4 平台,使用 ldr 指令即可将字节流每 4 字节加载到一个 32 位寄存器,节省了移位和拼接的操作.

由于 ARM Cortex-M4 寄存器资源有限,不能将待加载的 384 字节一次性加载到寄存器中. 考虑到数组指针、执行运算、暂存数据的需求,将 384 字节分为 12 次操作,也就是 12 次循环. 每个循环加载 32 个字节到 8 个寄存器,再依次对这 8 个寄存器做多项式采样的第二部分 CBD 函数处理,将结果存到相应的多项式位置,再进行下一次循环. 对这些寄存器做 CBD 函数的处理方式是相同的,故下述以加载字节流后的一个 32 位寄存器 Rt 为例.

首先,计算寄存器 Rt 中每 2 位的汉明重量,结

果存放在寄存器 Rd 的对应 2 位(使用算法 4 的第 2、3 行思想). 随后，取 Rd 中的每 4 位，高低 2 位做减法，减法结果即为采样所得的多项式系数. 这里的减法操作可以实现 4×8 位并行，是速度提升的关键点. 因为 ARM Cortex-M4 最小的数据处理单元为 8 位，而 $a,b\in\{0,1,2\}$，其相减结果不会超过 3 位，故可以先将 a 和 b 做 8 位有符号减法，再将结果扩展到 16 位，也就是多项式系数的数据类型.

如图 2 所示，对存放汉明重量的寄存器 Rd 做如下操作. ① 预处理：先对寄存器 Rd 进行移位操作，因为要对每 4 位的高低 2 位(a_i 和 b_i)做减法，对于寄存器 Rd 和其移位后存放的寄存器 Rd1、Rd2、Rd3，保留每 8 位的低 2 位，高 6 位为 0，从而将 2 位数据扩展为 8 位，这一过程通过与操作实现. 因此，寄存器 Rd 中的 16 个 2 位数扩展为 16 个 8 位数，为了执行下一步的减法并行，将同一索引的 a_i 和 b_i 放在不同寄存器的相同位置；② 减法并行：使用两次 ssub8 指令，每次对两个寄存器中的 4 对 8 位数据做有符号减法，结果存在寄存器 Rf0、Rf1 中，从而实现四并行；③ 扩展存放：先对寄存器 Rf0、Rf1 做移位操作，使得要扩展的 8 位数据放在寄存器每 16 位的低 8 位. 然后使用 sxtb16 指令将这些 8 位数据有符号展开为 16 位，也就是采样得到的多项式系数.

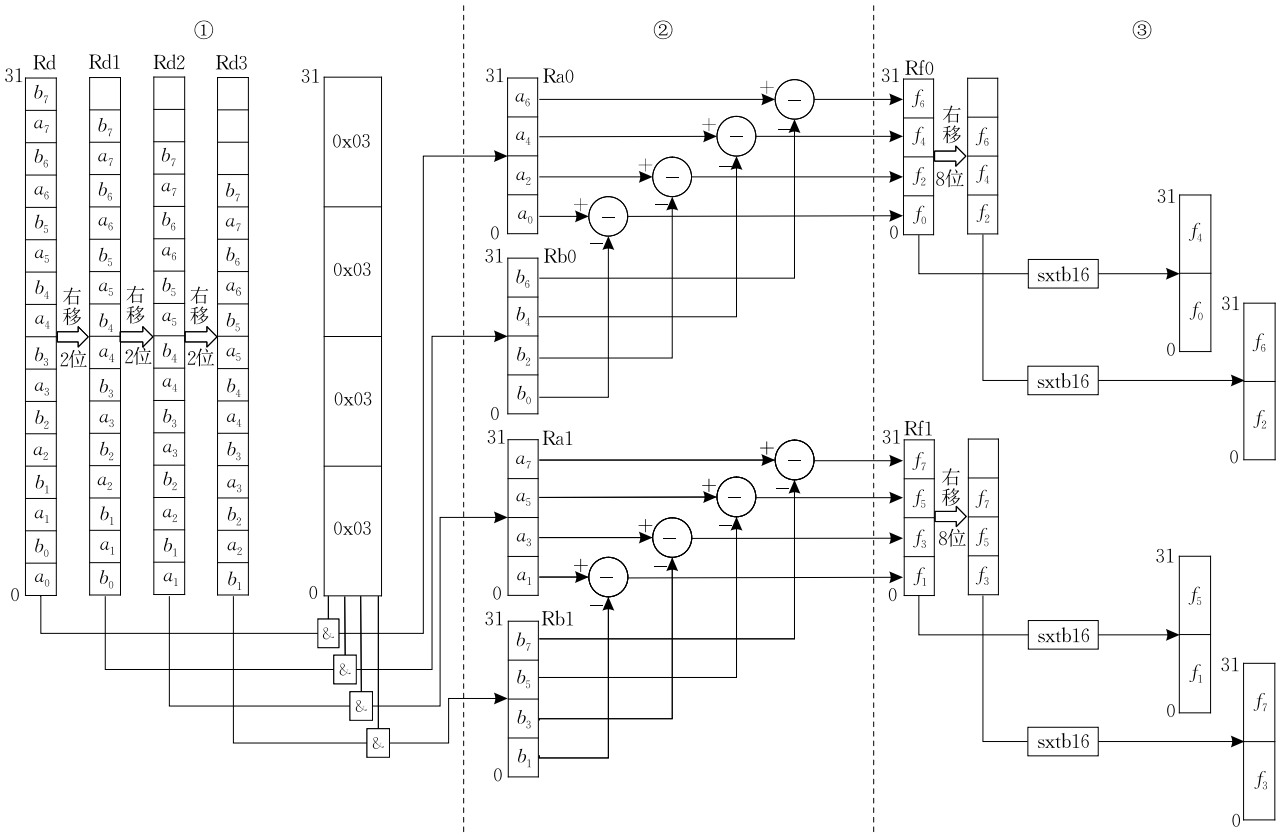


图 2　ARM Cortex-M4 并行实现 CBD_2

4.2　多项式快速数论变换

4.2.1　层融合

本文实现充分利用 NTT 运算层融合[41]的优化技巧. 层融合的思想是通过一次加载完成多层变换，减少内存加载和存储的次数，从而提升速度. 本文将多项式环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上的第 1 层分解和第 1～3 层基 2NTT 合并为四层融合，第 4～6 层基 2NTT 合并为三层融合，单独执行最后一层基 3NTT，即 4+3+1 层融合完成 8 层变换. 图 3 是正向变换的完整过程，每一数字块表示一个寄存器，每个寄存器存放两个 16 位系数，例如 0 表示该寄存器存放系数($a_1\|a_0$)，相当于二并行加速.

由于寄存器数量的限制，每次层融合分为多组循环进行，直到 768 个系数全部完成变换，每次循环加载的系数是由做蝴蝶变换的系数距离决定的. 具体展开前四层融合的第一次循环来说，加载第一组系数到通用寄存器 $r_2=(a_{49}\|a_{48})$，$r_3=(a_{145}\|a_{144})$，$r_4=(a_{241}\|a_{240})$，…，$r_9=(a_{721}\|a_{720})$，对其进行前三层变换，一对变换系数的距离分别为 384、192、96. 这部分系数恰好需要在第四层蝴蝶变换中和旋转因

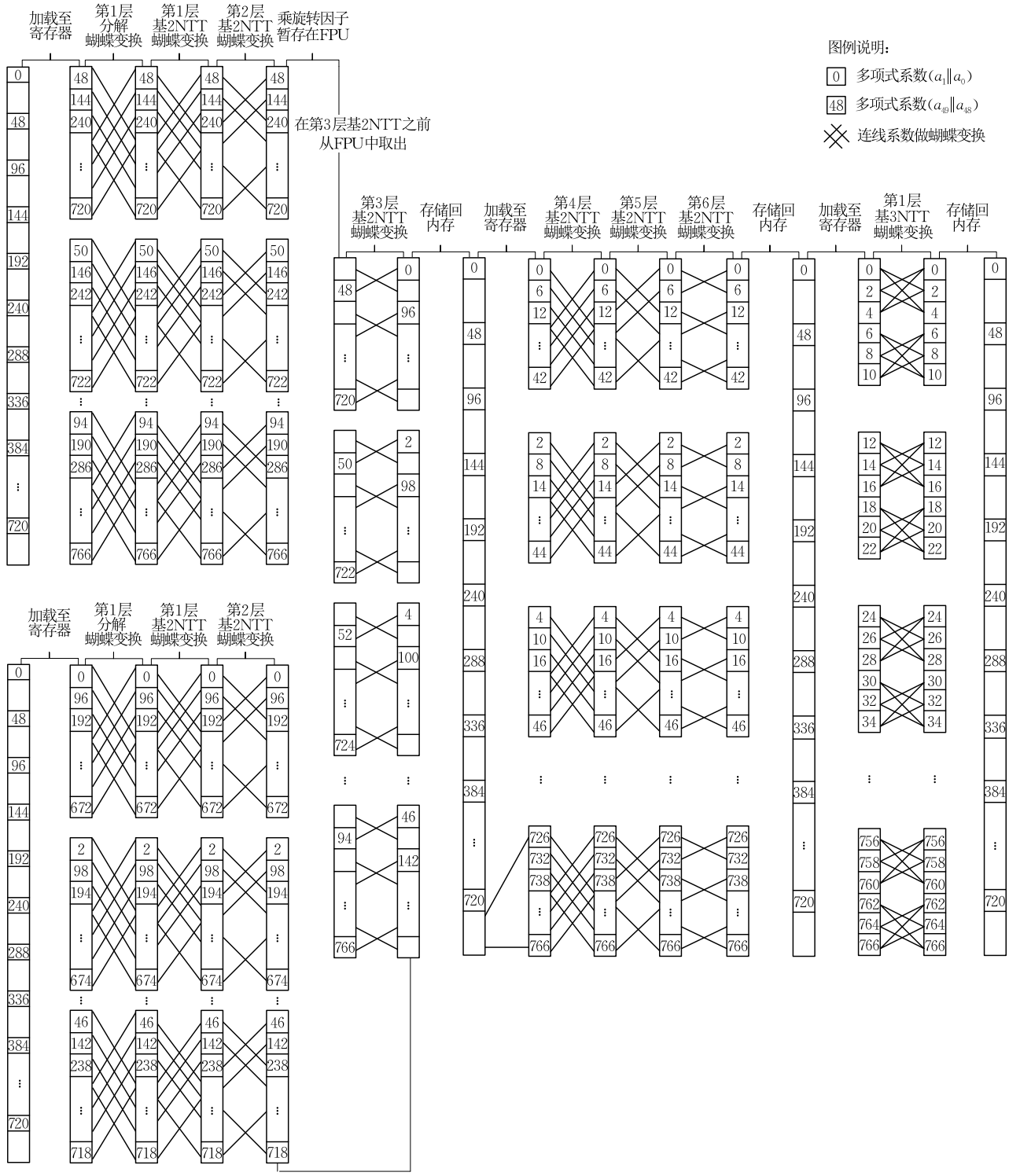


图 3 ARM Cortex-M4 实现混合基 NTT 层融合

子相乘,所以再将其乘以对应的旋转因子,结果暂存在 8 个 FPU 里,留作第四层使用. 加载第二组系数到通用寄存器 $r_2=(a_1\|a_0)$, $r_3=(a_{97}\|a_{96})$, $r_4=(a_{193}\|a_{192})$, …, $r_9=(a_{673}\|a_{672})$,对其进行前三层变换. 然后用 vmov 指令依次取出 FPU 里暂存的上一组变换乘积,和此时 r_2, r_3, …, r_9 里的系数做第四层变换,变换系数的距离间隔为 48. 因为 FPU 里的系数是前四层变换结果和旋转因子的乘积,第四层只需做加减操作,变换结束后存回原地址. 前四层融合的每次循环处理 32 个系数,共 24 次循环,每次循环结束后,地址指针前进 4 字节,进入下一次循环. 四层融合减少了 768×6 次内存加载和存储的操作,大

幅提升 NTT 运算效率.

基 2NTT 是每两个系数做变换，而基 3NTT 是三个系数为一组做变换，故不将基 3NTT 与基 2NTT 层融合. 第 4～6 层基 2NTT 变换系数的距离分别为 24、12、6，这三层融合共 16 次循环，每次循环处理 48 个系数. 以第一次循环为例，加载系数 $r_2=(a_1\|a_0)$，$r_3=(a_7\|a_6)$，$r_4=(a_{13}\|a_{12})$，…，$r_9=(a_{43}\|a_{42})$，对其进行 3 层变换，变换结束后存回原地址，地址指针前进 4 字节指向 a_2. 加载下一组系数完成同上操作，地址指针指向 a_4，再加载下一组系数完成 3 层变换后存回原地址，地址指针前进 88 字节指向 a_{48}，进入下一次循环. 如此系数分组进行循环是为了将 $(a_0,a_1,\cdots,a_{47})$ 在一组循环里完成变换，因为这部分系数对应的旋转因子是相同的. 每次循环加载旋转因子后即可更新其地址，下一次循环加载下一组系数对应的旋转因子. 三层融合减少了 768×4 次内存加载和存储的操作.

最后，单独执行系数距离为 2 的基 3NTT，得到正向 NTT 的最终结果.

4.2.2　循环展开

本文采用循环展开的方式，并以此迭代遍历每一层. 文献[42]和[25]需要空出一个寄存器存放循环计数器. 文献[42]使用循环计数器来检测循环次数，并用其确定使用哪一个旋转因子；文献[25]仅使用循环计数器检测结束循环的时间点. 本文为了节省寄存器的使用，不让循环技术占用多余的寄存器. 使用原始的. rept n 指令直接重复相同的代码块 n 次来实现循环的效果，而非使用寄存器进行循环计数. 虽然使用. rept 指令会增加代码尺寸，但是它直接重复代码的方式节省了加载循环计数寄存器的时间，也可以空出一个寄存器存储更多的系数，因而在速度上提升显著. 代码尺寸的增大对方案测试影响主要在烧写时间上，而算法性能主要关注算法运行所占用的时间和堆栈空间，不包括烧写时间，并且循环展开和寄存器计数两种方式占用的堆栈空间相同. CNTR 是紧凑型密钥封装方案，本身的代码体量小. rept 方式增加的代码量不会对方案性能和空间占用造成影响.

4.2.3　FPU

ARM Cortex-M4 仅有 14 个可用的通用目的寄存器，而在实际运算中，通用寄存器数量是紧俏的. 利用 FPU 存储利用率高的数据，使用 vmov 指令做通用寄存器和 FPU 之间的数据转换，从而减少内存加载和存储操作.

文献[32,35]证明使用 FPU 更加高效. 本文实现中，多项式环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 的第一层分解和前三层基 2NTT 需要 15 个旋转因子，将其打包进 8 个 FPU，在 24 次循环中重复使用. 使用 vldm 指令一次性加载旋转因子到 FPU，仅需要 9 个时钟周期，节省 24×8 个加载旋转因子到通用寄存器的操作. 然后，在每次循环中使用 vmov 指令将旋转因子从 FPU 取出，在通用寄存器中做相应的蝴蝶操作，每条 vmov 指令消耗一个时钟周期. 在后三层基 2NTT 运算时，由于 16 次循环中每次使用的旋转因子是不同的，并不会重复使用某个旋转因子，故使用 FPU 与直接从内存加载旋转因子的时间是相同的. 为统一代码形式，实现中依旧采用 FPU 暂存后三层基 2NTT 的旋转因子.

在混合基 NTT 的前四层融合时，使用 FPU 存储部分系数和旋转因子相乘的结果，便于在第四层调用. 同时，剩余的 FPU 也可以暂存中间值，处理通用寄存器紧缺的情况. 需要注意的是，运算操作只能在通用寄存器中进行，FPU 充当暂存数据的角色，并不能直接参与到运算指令中.

4.2.4　指令安排

ARM Cortex-M4 实现平台主要耗时在内存加载和存储操作，一条加载或存储指令需要 1 或 2 个时钟周期，如果单独执行 ldr 或 str 指令，一条指令耗时 2 个时钟周期；如果该条 ldr 或 str 指令是紧跟其他 ldr 或 str 指令执行，那么它需要消耗 1 个时钟周期. N 条连续的 ldr 或 str 指令需要 $N+1$ 个时钟周期. 为优化实现，将 ldr 或 str 指令放在一起连续执行，并且尽可能减少内存访问. 因此，对于加载进寄存器的系数，对其尽可能多地做运算，将 C 实现里分开运算的函数合并到一轮运算中再存储回原位置，以节省数据加载和存储的次数.

4.3　多项式求逆

令 $f=f_0+f_1x\in\mathbb{Z}_q/(x^2-\zeta)$，$g=g_0+g_1x\in\mathbb{Z}_q/(x^2-\zeta)$，多项式 $h=fg \bmod (x^2-\zeta)$，则 h 可以表示为 $h=h_0+h_1x$，其中 $h_0=f_0g_0+f_1g_1\zeta$，$h_1=f_1g_0+f_0g_1$. 同样的，多项式乘法 $h=fg \bmod (x^2-\zeta)$ 也可以用 $\mathbb{Z}_q$ 上的矩阵向量乘法表示，其中包含多项式 f 的旋转矩阵：

$$\begin{pmatrix}h_0\\h_1\end{pmatrix}=\begin{pmatrix}f_0 & f_1\zeta\\ f_1 & f_0\end{pmatrix}\begin{pmatrix}g_0\\g_1\end{pmatrix}.$$

为了计算多项式 $f \bmod (x^2-\zeta)$ 的逆，我们可以转而计算矩阵的逆，因此多项式 f 的逆存在的条件就是矩阵的逆存在. 如果矩阵行列式为 0，则多项

式不可逆.因此逆多项式对应的旋转矩阵也是多项式 f 旋转矩阵的逆矩阵.因此我们可以通过读取逆矩阵的第一列的系数从而得到多项式 f 的逆的系数.因此我们有:$f^{-1}=d^{-1}(f'_0+f'_1x)$,其中系数 f'_i 计算如下:

$$f'_0=f_0,$$
$$f'_1=-f_1,$$

其中 d 是多项式 f 的旋转矩阵的行列式,$d=f_0^2-f_1^2\zeta$.因此我们需要 10 个乘法和一个求 d 的逆元运算,由费马小定理可得:$d^{-1}=d^{q-2}=d^{3455}$.因此求逆元也需要额外的 20 个乘法.

对多项式求逆进行 ARM Cortex-M4 实现,主要难点在于计算行列式的逆,也就是 d^{q-2} 这一幂运算过程,我们将多项式求逆的过程封装成一个宏,每次可以计算四个系数也就是两个一次多项式的行列式求逆.并将两个多项式的行列式结果利用 pkhtb 指令打包到一个 32 位寄存器存回.在计算模幂结果时,我们利用汇编指令进行循环展开,减少了判断分支所需要的时间.

4.4 解密优化过程

在 CNTR 和 CTRU 的多项式解密算法中,含有 $h=f\cdot g\in\mathbb{Z}_{q_2}[x]/(x^n-x^{n/2}+1)$ 乘法运算,而 $q_2=1024$ 不是 NTT 友好素数,故不能直接使用 NTT.本文使用多模数 NTT 和 CRT 结合的方法完成 $\mathbb{Z}_{q_2}[x]/(x^n-x^{n/2}+1)$ 上的乘法运算.根据定理 1 使用 CRT 的条件,选取两个 NTT 友好素数:$q=3457$ 和 $q'=7681$,分别计算 f,g 在 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 和 $\mathbb{Z}_{q'}[x]/(x^n-x^{n/2}+1)$ 的变换,然后根据定理 1 重构运算得到在 $\mathbb{Z}_{q_2}[x]/(x^n-x^{n/2}+1)$ 上 $f\cdot g$ 的结果.

定理 1. 令 q_i 为正奇数,并且两两互素($gcd(q_i,q_j)=1$, $1\leq i<j<s$).如果 $|u_i|<q_i/2$,$|u|<\prod_{i=1}^{s}q_i$,那么 $u\equiv u_i(\mathrm{mod}\ q_i)$, $i=1,\cdots,s$ 的显式解 u 由下述式子可得:

$$\begin{cases} y_1=u_1 \\ y_2=y_1+((u_2-y_1)m_2 \bmod\pm q_2)q_1 \\ y_3=y_2+((u_3-y_2)m_3 \bmod\pm q_3)q_1q_2 \\ \quad\vdots\quad\vdots \\ u=y_s=y_{s-1}+((u_s-y_{s-1})m_s \bmod\pm q_s)q_1\cdots q_{s-1} \end{cases},$$

其中,$m_i:=(q_1\cdots q_{i-1})^{-1}\bmod\pm q_i$.

环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上的 NTT 可以复用 4.2 节介绍的 NTT.环 $\mathbb{Z}_{q'}[x]/(x^n-x^{n/2}+1)$ 是 NTTRU 方案使用的多项式环,同时 NTTRU 有 ARM Cortex-M4 的实现,因此在实现过程中这部分代码复用文献[37]的实现.

4.5 模约减

4.5.1 模约减算法

模约减操作 $c \bmod q$ 可以通过 $c-\lfloor c/q\rfloor\cdot q$ 完成,其中除法操作不能在常数时间内实现.由于安全性问题,非常数时间的除法操作不能用于密码算法中,一般使用常数时间的模约减 Barrett 算法[29]和 Montgomery 算法[27],以及文献[37]中使用 Plantard 算法[38].Plantard 算法适用于 16×32 位乘法,而本文使用的数据都在 16 位有符号整数范围内,使用 Plantard 算法前需要预处理数据,例如将 16 位的旋转因子预计算到 32 位,有较大的存储空间消耗.Barrett 算法直接约减 16 位有符号整数,Montgomery 算法将 16×16 位的乘法结果约减到模 q 域内,为了节省空间的消耗,本文实现主要使用 Barrett 算法和 Montgomery 算法.

算法 5 描述了 Barrett 约减在 ARM Cortex-M4 上的实现,一次调用可以完成 2 个 16 位有符号数的约减.这是一种改进后的 Barrett 算法,比文献[26]使用的 Barrett 算法减少 2 条指令.ARM Cortex-M4 平台多数指令执行时间为一个时钟周期,将原本乘法加法两条指令替换为 smlaw{b,t}一条指令,从而节省一个时钟周期的消耗,显著提升运算速度,但是需要多 1 个寄存器存储约减常量.而在 CNTR 中,对单个 768 维多项式调用 Barrett 约减,用 1 个寄存器的空间换 1536 条指令的时间,可以达到时间和空间的平衡,从而提升约减效率.

算法 5. Barrett 约减算法 Cortex-M4 实现.

输入:封装 2 个系数的 32 位寄存器单元 $a=(a_t\|a_b)$

输出:约减后的 32 位寄存器单元 $c=(a_t\|a_b)\bmod\pm q$

1. $v_1\leftarrow-\lfloor 2^{32}/q\rceil$ ▷预计算 v_1
2. $v_2\leftarrow 2^{15}$ ▷预计算 v_2
3. smlawb t_0,v,a,v_2 ▷$t_0\leftarrow(v_1\cdot a_{\mathrm{bottom}}+2^{31})/2^{16}$
4. smlabt t_0,q,t_0,a ▷$t_0\leftarrow q\cdot t_{0\,top}+a$
5. smlawt t_1,v,a,v_2 ▷$t_1\leftarrow(v_1\cdot a_{\mathrm{top}}+2^{31})/2^{16}$
6. smulbt t_1,q,t_1 ▷$t_1\leftarrow q\cdot t_{1_{\mathrm{top}}}$
7. add $t_1,a,t_1,lsl\#16$ ▷$t_1\leftarrow a+(t_1\ll 16)$
8. pkhtb c,t_0,t_1 ▷$c\leftarrow(t_{0_{\mathrm{top}}}\|t_{1_{\mathrm{bottom}}})$

算法 5 的输入为封装在 32 位寄存器单元的 2 个 16 位有符号数 a_t 和 a_b,输出为 Barrett 约减后的对应结果.v_1 和 v_2 是写入寄存器的预计算常量,不需要在每次运行 Barrett 约减算法都计算,从而不占用该算法的时间消耗.

本文应用了改进后的 Montgomery 约减,见算

法 6. 算法的输入为两个 16 位有符号数的乘积. 输出的高 16 位为 Montgomery 约减后的结果. 将预存储的 q^{-1} 调整为 $-q^{-1}$，则原本 Montgomery 算法中涉及 q^{-1} 的减法可调整为 $-q^{-1}$ 的加法，使用 smlabb 指令在一个时钟周期内完成 2 个 16 位的乘积和一个 32 位的加法.

算法 6. Montgomery 约减算法 Cortex-M4 实现；使用 Montgomery 因子 $\beta=2^{16}$.

输入：32 位寄存器单元 a，其中 $-\frac{\beta}{2}q\leq a<\frac{\beta}{2}q$

输出：约减后的 32 位寄存器单元 r，其中 $r_{\text{top}}=\beta^{-1}a(\bmod q)$

1. smulbb $t,a,-q^{-1}$　　$\triangleright t\leftarrow(a \bmod \beta)\cdot(-q^{-1})$
2. smlabb r,t,q,a　　$\triangleright r_{\text{top}}\leftarrow\left\lfloor\frac{(t \bmod \beta)\cdot q}{2^{16}}\right\rfloor+\left\lfloor\frac{a}{2^{16}}\right\rfloor$

算法 7. 封装 2 个系数的 Montgomery 约减算法 Cortex-M4 实现；使用 Montgomery 因子 $\beta=2^{16}$.

输入：封装 2 个系数的 32 位寄存器单元 $a=(a_t\|a_b)$

输出：约减后的 32 位寄存器单元 r，其中 $r\equiv(a_t\|a_b) \bmod q$

1. $v\leftarrow\beta(\bmod q)$　　$\triangleright$预计算
2. smulbb t_1,a,v　　$\triangleright t_1\leftarrow a_{\text{bottom}}\cdot v$
3. smulbb $r_1,t_1,-q^{-1}$　　$\triangleright r_1\leftarrow(t_1 \bmod \beta)\cdot(-q^{-1})$
4. smlabb r_1,r_1,q,t_1　　$\triangleright r_{1\,\text{top}}\leftarrow\left\lfloor\frac{(r_1 \bmod \beta)\cdot q}{2^{16}}\right\rfloor+\left\lfloor\frac{r_1}{2^{16}}\right\rfloor$
5. smultb t_2,a,v　　$\triangleright t_2\leftarrow a_{\text{top}}\cdot v$
6. smulbb $r_2,t_2,-q^{-1}$　　$\triangleright r_2\leftarrow(t_2 \bmod \beta)\cdot(-q^{-1})$
7. smlabb r_2,r_2,q,t_2　　$\triangleright r_{2\,\text{top}}\leftarrow\left\lfloor\frac{(r_2 \bmod \beta)\cdot q}{2^{16}}\right\rfloor+\left\lfloor\frac{r_2}{2^{16}}\right\rfloor$
8. pkhtb $r,r_2,r_1,asr\ \#16$　　$\triangleright r\leftarrow(r_{2\,\text{top}}\|(r_{1\,\text{top}}\geq 16))$

算法 6 是对乘积结果的 Montgomery 约减，输出是在 F_q 域内 β^{-1} 和输入的乘积. 如果对 2 个 16 位有符号数分别做 Montgomery 约减，则使用算法 7. 算法 7 的输入是 2 个 16 位的有符号数 a_t 和 a_b，输出为打包好的 2 个 16 位模 q 约减结果. 预计算 $\beta=2^{16}(\bmod q)$ 和 16 位有符号数相乘，得到和 Barrett 约减相同的结果.

4.5.2　延迟约减

我们采用最接近模数 3457 的最小计算机字长 16 位存放多项式系数，16 位有符号数范围为[−32768，32767]，因此在进行正向 NTT 并不需要每层对所有系数进行约减，只要保证系数范围不溢出 16 位即可. 在 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上正向 NTT 运算的输入范围有三种情况. 密钥生成阶段对多项式 f 和 g 做 NTT，多项式 f 由 CBD_2 生成再扩为 2 倍，首位加 1，故 f 的系数范围是[−4，5]，g 由 CBD_2 生成，其系数范围是[−2，2]. 加密过程对多项式 r 做 NTT，r 由 CBD_2 生成，其系数范围是[−2，2]. 解密过程对密文多项式 c 和私钥多项式 f 做 NTT，c 的系数范围是[0，1024)，f 的范围和密钥生成阶段的 f 是相同的. 以下针对正向 NTT 输入范围的三种情况分析 Montgomery 约减和 Barrett 约减的延迟操作.

输入正向 NTT 的多项式系数范围是[−4，5]时，NTT 中使用的旋转因子在 $(-q,q)$ 内，所以第一层分解中系数和旋转因子的乘积不溢出 16 位有符号数，具体范围是[−4×723，5×723]，该乘法结果不用做 Montgomery 约减. 多项式环 $f \bmod (x^n-x^{n/2}+1)$ 分解为 $f \bmod (x^{n/2}-\zeta_1)$ 和 $f \bmod (x^{n/2}-\zeta_2)$ 两个环的计算后，系数增长到[−4×723+(−4)×2，5×723+5×2]，此时加减也不需要做 Barrett 约减. 在正向基 2NTT 过程中，如果只有乘法做 Montgomery 约减，加减法不做约减，则第一层基 2NTT 之后系数的范围是 $(-q-4\times725,q+5\times725)$，第二层增长到 $(-2q-4\times725,2q+5\times725)$. 直到最后一层基 2NTT，系数范围为 $(-6q-4\times725,6q+5\times725)$，综上分析基 2NTT 六层结束后不会溢出 16 位有符号数，六层基 2NTT 加减不需要做 Barrett 约减. 在最后一层基 3NTT，输入系数范围为 $(-6q-4\times725,6q+5\times725)$，加减后系数范围将增长到 $(-8q-4\times725,8q+5\times725)$，不溢出 16 位有符号数，因此基 3NTT 结束后也不需要进行延迟约减. 采用延迟约减，在正向 NTT 时，总共减少了 384 次 Montgomery 约减和 768×8 次 Barrett 约减，提升多项式乘法运行效率.

其他两种系数范围的分析和上述一致，多项式系数范围是[−2，2]时和系数范围为[−4，5]时的约减次数相同. 多项式系数范围是[0，1024)时，所有乘法结果都做 Montgomery 约减，只在最后一层基 2NTT 做 Barrett 约减，减少 768×7 次 Barrett 约减.

$\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上逆向 NTT 运算的输入范围是 $(-q,q)$，使用和正向 NTT 相同的延迟约减分析方法. 具体分析每一层变换的系数范围，得出在逆向变换时，所有乘法结果都做 Montgomery 约减，逆向基 2NTT 的第一和四层的加法做 Barrett 约减，共做 384×2 次 Barrett 约减，减少了 384×6 次 Barrett 约减.

4.6　栈空间优化

嵌入式设备的存储空间是主要瓶颈，算法所需的空间很大程度影响着实际部署算法的可行性. 考虑到这一因素，本文在尽可能不影响性能的情况下，

减少堆栈空间消耗，主要使用空间复用的优化技巧. 本文对CNTR和CTRU方案的ARM Cortex-M4优化实现以堆栈使用和速度为主要指标，同时保持代码大小的合理性.

在CNTR和CTRU的C参考实现中，对方案所需的所有变量都申请了空间，并且分别依次对变量做操作. 由于运算的线性特性，以及某些元素之间的独立性，可以使用同一多项式变量完成多个独立的操作. 具体来说，在本文CNTR实现中，密钥生成阶段申请了3个多项式变量，比原始实现减少了2个变量；密钥封装阶段申请了2个多项式变量，比原始实现减少了2个变量；在密钥解封装阶段，由于使用4.4节介绍的优化方法，比原始实现多申请了2个多项式变量，而在速度上有明显提升，权衡时间和空间的平衡性，这样的消耗是值得的. 在本文CTRU实现中，密钥生成和密钥解封装阶段与CNTR申请的多项式变量数量一致，密钥封装阶段比CNTR多申请1个多项式变量，用于存放噪声多项式 e. 每个多项式的运算都在申请的空间上进行，最后存入相应的密钥和密文位置.

4.7 抗侧信道攻击分析

本文对CNTR和CTRU的实现是常数时间的. 由于多项式系数在 $\mathbb{Z}_q$ 域内，实现中需要频繁对数据进行取模运算. 一般的取模运算(%)本质是除法运算，不是常数时间，容易受到侧信道攻击. 本文使用Barrett算法和Montgomery算法进行模约减，其均为常数时间算法，并使用DSP指令集进一步优化算法以提升效率. DSP指令集中一条指令一般占1个时钟周期，算法在常数时间运行，能够抵抗侧信道攻击. 此外，约减算法并不局限于特定的 q，不会暴露环的模数. E_8 格加解密算法也是常数时间实现的，并且不依赖于秘密信息的分支，所以能够抵抗侧信道攻击.

5 实验结果

5.1 测试环境

本文的测试平台是带有ARMv7E-M指令集的STM32F4DISCOVERY开发板，其内存为196 KB，闪存为1 MB. 测试的硬件环境为2.50 GHz Intel(R) Core(TM) i5-12400F处理器和32 GB内存，软件环境为Windows11操作系统和wsl2子系统，编译环境为带有-O3选项的arm-none-eabi-gcc 9.2.1版本. 基于pqm4[①]搭建基准测试框架，所有算法都在相同条件下运行和测试，以比较算法性能. 算法中涉及的哈希函数也是统一使用pqm4提供的标准.

由于本文工作是CNTR和CTRU方案的首次ARM Cortex-M4实现，故方案的优化效果和C参考实现比较得出. 并且，比较其他NTRU格基密钥封装方案的最新ARM Cortex-M4实现[33,35,37]，NTRU-HRSS[②]、SNTRU Prime-761[③]、NTTRU[④]的性能数据均使用其团队的开源代码. 在和基于{R,M}LWE/LWR问题的方案比较时，选取NIST拟标准化的密钥封装方案Kyber的最新实现[23]，源码从pqm4获取. 为了进行公平的比较分析，将以上算法在相同的测试环境下运行，以算法运行的时钟周期数和堆栈使用情况分别作为时间和空间的性能指标.

5.2 多项式运算优化测试

多项式运算是密码算法中最为常见的核心操作之一，其中，多项式乘法是耗时最长的操作，因此成为工程实现中主要的优化目标. 为了提高多项式运算的效率，本文使用汇编指令对CNTR和CTRU方案进行优化.

如表1所示，相较于文献[19]的C参考实现，本文的优化方案在多项式中心二项分布采样、多项式求逆上分别提升了32.49%、68.85%. CNTR和CTRU中环 $\mathbb{Z}_q[x]/(x^n-x^{n/2}+1)$ 上的多项式乘法使用混合基NTT，运用层融合、循环展开、延迟约减等技术，将NTT、逆向NTT、点乘的速度分别提升84.24%、81.15%、87.36%，从而将该环上多项式乘法速度整体上提升83.65%. 针对解密过程中非NTT友好素数模数的多项式环乘法，使用多模数NTT与CRT结合的方法计算多项式乘法，将解密过程的速度提升96.26%. 对这些多项式运算的关键步骤进行优化，性能取得了显著提升.

表1 CNTR/CTRU优化模块前后多项式运算速度对比

运算模块	优化前/时钟周期	优化后/时钟周期	优化率/%
多项式采样	4106	2772	32.49
多项式求逆	135242	42129	68.85
NTT	120986	19063	84.24
逆向NTT	135053	25454	81.15
点乘	47689	6029	87.36
多项式乘法	424963	69475	83.65
解密过程	5377957	201111	96.26

循环是多项式运算中重要的环节，采用第4.2.2

① https://github.com/mupq/pqm4
② https://github.com/iremkp/NTRU-tmvp4-m4
③ https://github.com/vincentvbh/NTRUPrime-PolyMul
④ https://github.com/UIC-ESLAS/ImprovedPlantardArithmetic

节介绍的循环展开技术，优化数据如表 2 所示，优化模块的速度提升率在 2%～18%之间. 其中，点乘运算循环判断之间的代码较少，如果采用寄存器计数方式会频繁调用循环计数器，使用循环展开技术可以大幅减少寄存器调用，故提升效果最为明显.

表 2　使用循环展开技术前后模块对比

运算模块	使用循环展开前的速度/时钟周期	使用循环展开后的速度/时钟周期	优化率/%
多项式采样	2856	2772	2.94
多项式求逆	43090	42129	2.23
NTT	19459	19063	2.04
逆向 NTT	27481	25454	7.38
点乘	7372	6029	18.22
多项式乘法	73638	69475	5.65
解密过程	205257	201111	2.02

如表 3 所示，相较于其他 NTRU 格基密钥封装方案的 ARM Cortex-M4 实现，本文实现的 CNTR 和 CTRU 在多项式乘法方面表现出明显的速度优势，尤其是相较于 NTRU-HRSS[33] 和 SNTRU Prime-761[35] 而言. 虽然我们的正向 NTT 和逆向 NTT 速度略慢于 NTTRU[37]，但我们的点乘速度快于 NTTRU. 此外，NTTRU[37] 中使用的 Plantard 约减方法需要占用更多的空间，而本文实现在时间和空间的权衡上更具优势，从而在效率方面表现出一定的优越性. 由于 Kyber 算法中的多项式维度远小于 CNTR 和 CTRU，并且 Kyber 中使用 7 层基 2NTT 完成 NTT 友好环的多项式乘法，而 CNTR/CTRU 使用混合基 NTT 完成非 NTT 友好环多项式乘法运算，所以 Kyber 中多项式乘法速度快于本文实现，但是 Kyber 在高维情况下的安全性更为脆弱，低于 CNTR 和 CTRU.

表 3　本文和其他格基密钥封装方案 ARM Cortex-M4 实现的多项式运算速度对比　（单位：时钟周期）

方案	环	正向 NTT	逆向 NTT	点乘	多项式乘法	多项式求逆
CNTR/CTRU	$\mathbb{Z}_{3457}[x]/(x^{768}-x^{384}+1)$	19063	25454	6029	69475	42129
NTRU-HRSS[33]	$\mathbb{Z}_{8192}[x]/(x^{701}-1)$	—	—	—	142030	—
SNTRU Prime-761[35]	$\mathbb{Z}_{4591}[x]/(x^{761}-x-1)$	36857	47040	25641	148168	—
NTTRU[37]	$\mathbb{Z}_{7681}[x]/(x^{768}-x^{384}+1)$	17287	20960	10577	65980	40788
Kyber-768[23]	$\mathbb{Z}_{3329}[x]/(x^{256}+1)$	4486	4665	1228	14730	—

5.3　方案整体性能测试

密钥封装方案的整体性能分析中，关注密钥生成、密钥封装和密钥解封装三个阶段的速度和堆栈使用情况.

5.3.1　与 C 参考实现比较

表 4 展示了本文实现的 CNTR 和 CTRU 方案相较于文献[19]中 C 参考实现的性能提升. 在速度方面，CNTR 的密钥生成、密钥封装、密钥解封装分别提升 59.17%、58.42%、90.26%；CTRU 的密钥生成、密钥封装、密钥解封装分别提升 62.81%、56.85%、90.02%；CNTR 和 CTRU 的整体速度优化率分别为 85.54%、85.56%. CNTR 和 CTRU 的显著加速主要来自 NTT、逆向 NTT、点乘、多项式采样和多项式求逆的优化汇编实现，并使用多模数 NTT 改进解密过程的非 NTT 友好素数模数的多项式环乘法，使得密钥解封装速度提升最为明显，从而提升算法的整体速度. 在堆栈空间方面，CNTR 的密钥生成、密钥封装、密钥解封装分别提升 34.27%、43.76%、15.71%；CTRU 的密钥生成、密钥封装、密钥解封装分别提升 35.81%、35.12%、15.40%；CNTR 和 CTRU 的整体空间优化率分别为 29.86%、28.17%. CTRU 的密钥封装阶段堆栈空间明显多于 CNTR 的原因在于 CTRU 增加了噪声多项式 e，需要多申请一个多项式变量. 以上数据表明，本文的优化技术可以大幅度提高 CNTR 和 CTRU 的实现效率.

表 4　CNTR/CTRU 优化前后速度和堆栈空间对比

方案	阶段	速度/时钟周期			堆栈空间/字节		
		优化前	优化后	优化率/%	优化前	优化后	优化率/%
CNTR	密钥生成	600 k	245 k	59.17	8928	5868	34.27
	密钥封装	469 k	195 k	58.42	7020	3948	43.76
	密钥解封装	6040 k	588 k	90.26	9676	8156	15.71
	总计	7109 k	1028 k	85.54	25624	17972	29.86
CTRU	密钥生成	570 k	212 k	62.81	8544	5484	35.81
	密钥封装	489 k	211 k	56.85	8748	5676	35.12
	密钥解封装	6060 k	605 k	90.02	9868	8348	15.40
	总计	7119 k	1028 k	85.56	27160	19508	28.17

5.3.2 与其他格基密钥封装方案的比较

CNTR 和 CTRU 与其他密钥封装方案的比较如表 5 所示，从安全性、性能和空间使用几个角度进行对比分析. 本文实现的 CNTR/CTRU 参数集(n=768)对应 NIST 推荐的安全级别 III，对比方案也选择安全级别 III 参数集，以保证比较的公平性. 在表 5 中，“安全性假设”一列表示方案底层的困难性问题，“安全性规约”一列表示 IND-CCA 安全规约到哪种 CPA 安全，其中 IND 和 OW 分别表示不可区分性和单向性，RPKE 和 DPKE 分别表示随机化公钥加密和确定性公钥加密. 方案的安全性假设和安全性规约数据来源于文献[19].

表 5 本文和格基密钥封装方案 ARM Cortex-M4 实现速度和堆栈空间对比

方案	安全性假设	安全性规约	速度/时钟周期			堆栈空间/字节		
			密钥生成	密钥封装	密钥解封装	密钥生成	密钥封装	密钥解封装
CNTR	NTRU，RLWR	IND-CPA，RPKE	245 k	195 k	588 k	5868	3948	8156
CTRU	NTRU，RLWE	IND-CPA，RPKE	212 k	211 k	605 k	5484	5676	8348
NTRU-HRSS[33]	NTRU	OW-CPA，DPKE	151732 k	370 k	781 k	24816	16600	17808
SNTRU Prime-761[35]	NTRU	OW-CPA，DPKE	10753 k	679 k	564 k	61468	13336	16968
Kyber-768[23]	MLWE	IND-CPA，RPKE	600 k	620 k	782 k	5336	6456	7624
CNTR，SHA-2 变体	NTRU，RLWR	IND-CPA，RPKE	296 k	201 k	481 k	5868	3892	8092
CTRU，SHA-2 变体	NTRU，RLWE	IND-CPA，RPKE	244 k	228 k	508 k	5484	5620	8284
NTTRU[37]	NTRU	OW-CPA，RPKE	284 k	252 k	272 k	7084	6980	8284

NTRU-HRSS 和SNTRU Prime-761 都具有严格的开始于 OW-CPA 确定性公钥加密的 CCA 规约界限，但是解密过程更加复杂且耗时. 在任何情况下，IND-CPA 安全都是比 OW-CPA 安全更强的安全概念. 所以 CNTR/CTRU 的安全性高于 NTRU-HRSS和SNTRU Prime-761. 在性能方面，与NTRU-HRSS[33]对比，本文实现的 CNTR 方案在密钥生成阶段的速度快 619.31 倍，密钥封装阶段快 1.90 倍，密钥解封装阶段快 1.33 倍；CTRU 方案在密钥生成阶段的速度快 715.72 倍，密钥封装阶段快 1.75 倍，密钥解封装阶段快 1.29 倍. 由于其他方案也都在 ARM Cortex-M4 平台上完成了性能优化，故速度优势不如文献[19]中的 C 实现明显，但 CNTR 和 CTRU 仍然优于 NTRU-HRSS 的性能表现，并且在堆栈空间方面具有明显的优势. 与 SNTRU Prime-761[35]比较，CNTR 和 CTRU 在密钥解封装阶段略慢，因为 E_8 格解密算法更复杂、安全性更高，综合速度和堆栈空间两方面来看，本文实现是优于 SNTRU Prime-761 的.

为了与NTTRU[37]进行公平的比较，将CNTR/CTRU 做以下修改：(1)将哈希函数由 SHA-3 修改为 SHA-2；(2)使用 AES 加密算法扩展种子，而不使用 SHAKE；(3)将 FO 转换变为 $FO_m^{\perp}$，而不使用前缀哈希. NTTRU 的CCA安全规约到OW-CPA 安全，而CNTR/CTRU的 CCA 安全规约到 IND-CPA 安全，故CNTR/CTRU安全性强于 NTTRU. CNTR 和 CTRU 的 SHA-2 变体方案在密钥生成方面速度接近 NTTRU，密钥封装快于 NTTRU，但是密钥解封装慢于 NTTRU. 主要原因有以下两点：(1)为了实现更高的安全性，CNTR 和 CTRU 使用尺度化 E_8 格解密算法，比 NTTRU 的解密算法更为复杂，因而占用更多的时间；(2)CNTR 和 CTRU 在解密过程中使用的多模数 NTT 比 NTTRU 的 NTT 更耗时，因为多模数 NTT 本质上包含两个 NTT 算法的流程. 在堆栈空间方面，CNTR 和 CTRU 相较于 NTTRU 是具有明显优势的.

文献[43]没有提供替换 FO 转换的 Kyber 算法 Cortex-M4 开源代码，我们对 pqm4 中 Kyber 最新 Cortex-M4 实现代码进行修改，将 KEM 中的 FO 转换替换为 $FO_{ID(pk),m}^{\not\perp}$ 进行修改测试. 修改前后的 Kyber-768 Cortex-M4 实现测试数据变化规律与文献[43]的 AVX2 数据变化规律一致. 因此我们采用修改 FO 转换的 Kyber-768 Cortex-M4 代码进行测试，与相同 FO 转换构造下的 CNTR/CTRU Cortex-M4 实现进行比较. 在安全性方面，CNTR/CTRU 和 Kyber 的安全规约一致，错误率低于 Kyber[19]. 在性能方面，本文实现的 CNTR 方案在密钥生成、密钥封装、密钥解封装阶段的速度分别比 Kyber 快 2.45 倍、3.18 倍、1.33 倍；CTRU 方案在密钥生成、密钥封装、密钥解封装阶段的速度分别比 Kyber 快 2.83 倍、2.94 倍、1.29 倍. 因为在 Kyber 中，使用拒绝采样的方式生成矩阵，并且密钥生成和加密过程中含有复杂的“矩阵-向量”乘法；而 CNTR 和 CTRU 的加密过程仅有一个多项式乘法.

综合上述分析可以得出，CNTR 和 CTRU 方案在 ARM Cortex-M4 平台上的优化实现取得了显著

的效果.相较于其他格基密钥封装方案,CNTR 和 CTRU 在速度、空间和安全性方面具有综合性的优势.

6　总结与展望

本文针对 NTRU 格基密钥封装方案 CNTR 和 CTRU 完成嵌入式平台 ARM Cortex-M4 的优化实现,在算法速度和堆栈空间两方面实现算法性能的提升.加速多项式中心二项分布采样、混合基 NTT、多项式求逆、解密等核心步骤,将 CNTR 的密钥生成、封装、解封装速度分别提升 59.17%、58.42%、90.26%;CTRU 这三阶段的速度分别提升 62.81%、56.85%、90.02%.同时,大幅减少堆栈空间的消耗.此外,与其他密钥封装方案进行了比较和分析,结果表明本文实现在速度、空间和安全性方面具有综合性的优势.

CNTR 和 CTRU 目前已在我国密标委立项,对我国丰富抗量子格基密码技术路线具有实际意义.考虑到 Kyber 的专利风险和 CNTR/CTRU 的综合性能等因素,CNTR 和 CTRU 在嵌入式平台 ARM Cortex-M4 上的优化实现为更好地满足物联网安全需求提供了可行性和参考,对推进我国后量子密码算法的工程化和实用化具有较好的促进作用.

致　谢　本文评审专家提供了非常有帮助的评阅意见;梁志闯、李文倩等对本文写作和讨论给与了帮助和支持,在此一并感谢!

参　考　文　献

[1] Shor P W. Algorithms for quantum computation: Discrete logarithms and factoring//Proceedings of the 35th Annual Symposium on Foundations of Computer Science. Santa Fe, USA, 1994: 124-134

[2] NIST. Post Quantum Cryptography Round 3 Submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

[3] NIST. PQC standardization process: Announcing four candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4

[4] Regev O. On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM, 2009, 56(6): 1-40

[5] Lyubashevsky V, Peikert C, Regev O. On ideal lattices and learning with errors over rings. Journal of the ACM, 2013, 60(6): 1-35

[6] Langlois A, Stehlé D. Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography, 2015, 75(3): 565-599

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

[8] Alperin-Sheriff J, Apon D. Dimension-preserving reductions from LWE to LWR. Cryptology ePrint Archive, 2016: 589. https://eprint.iacr.org/2016/589

[9] Hoffstein J, Pipher J, Silverman J H. NTRU: A ring-based public key cryptosystem//Proceedings of the Algorithmic Number Theory: Third International Symposiun. ANTS-III Portland, Oregon, USA, 1998: 267-288

[10] Howgrave-Graham N, Silverman J H, Whyte W. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3//Proceedings of the Topics in Cryptology—CT-RSA 2005: The Cryptographers' Track at the RSA Conference 2005. San Francisco, USA, 2005: 118-135

[11] Howgrave-Graham N, Silverman J H, Singer A, et al. NAEP: Provable security in the presence of decryption failures. Cryptology ePrint Archive, 2003: 172. https://eprint.iacr.org/2003/172

[12] Prest T, Fouque P-A, Hoffstein J, et al. Falcon: Fast-fourier lattice-based compact signatures over NTRU specification v1.2. Submission to round 3 of the NIST post-quantum project, 2020. https://falcon-sign.info/

[13] Chen C, Danba O, Hoffstein J, et al. NTRU algorithm specifications and supporting documentation. Submission to round 3 of the NIST post-quantum project, 2020. https://ntru.org/

[14] Bernstein D J, Brumley B B, Chen M-S, et al. NTRU prime: Round 3. Submission to round 3 of the NIST post-quantum project, 2020. https://ntruprime.cr.yp.to/

[15] Avanzi R, Bos J, Ducas L, et al. CRYSTALS-kyber algorithm specifications and supporting documentation (Version 3.02). Submission to round 3 of the NIST post-quantum project, 2021. https://pq-crystals.org/

[16] Lyubashevsky V, Seiler G. NTTRU: Truly fast NTRU using NTT. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019, (3): 180-201

[17] Duman J, Hövelmanns K, Kiltz E, et al. A thorough treatment of highly-efficient NTRU instantiations//Public Key Cryptography (PKC) 2023: The 26th IACR International Conference on Practice and Theory of Public-Key Cryptography. Atlanta, USA, 2023: 65-94

[18] Stehlé D, Steinfeld R. Making NTRU as secure as worst-case problems over ideal lattices//Advances in Cryptology—

EUROCRYPT 2011: The 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques. Tallinn, Estonia, 2011: 27-47

[19] Liang Z, Fang B, Zheng J, et al. Compact and efficient KEMs over NTRU lattices. Cryptology ePrint Archive, 2022: 579. https://eprint.iacr.org/2022/579

[20] Cryptography Standardization Technical Committee (CSTC). 2023 annual encryption industry standard formulation and revision tasks (commercial encryption field). Beijing: CSTC, 2023(in Chinese)
(密码行业标准化技术委员会. 2023 年度密码行业标准制修订任务(商用密码领域). 北京: 密码行业标准化技术委员会, 2023)

[21] Zhang J, Yu Y, Fan S, et al. Supporting documentation of Aigis-enc. https://sfjs.cacrnet.org.cn/site/content/364.html

[22] Maayan G D. The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx

[23] Kannwischer M J, Rijneveld J, Schwabe P, et al. Pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4. Cryptology ePrint Archive, 2019: 844. https://eprint.iacr.org/2019/844

[24] D'Anvers J-P, Karmakar A, Roy S S, et al. SABER: Mod-LWR based KEM (Round 3 Submission). Submission to round 3 of the NIST post-quantum project. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/

[25] Botros L, Kannwischer M J, Schwabe P. Memory-efficient high-speed implementation of kyber on cortex-m4//Progress in Cryptology—AFRICACRYPT 2019: The 11th International Conference on Cryptology in Africa. Rabat, Morocco, 2019: 209-228

[26] Alkim E, Bilgin Y A, Cenk M, et al. Cortex-m4 optimizations for {R, M} LWE schemes. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020, (3): 336-357

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

[28] Abdulrahman A, Hwang V, Kannwischer M J, et al. Faster kyber and dilithium on the cortex-m4//Applied Cryptography and Network Security: The 20th International Conference, ACNS 2022. Rome, Italy, 2022: 853-871

[29] Barrett P. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor//Advances in Cryptology—CRYPTO'86: Proceedings. Berlin, Germany: Springer, 2000: 311-323

[30] Chung C M 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, (2): 159-188

[31] Toom A L. The complexity of a scheme of functional elements realizing the multiplication of integers. Soviet Mathematics Doklady, 1963, 3(4): 714-716

[32] Abdulrahman A, Chen J P, Chen Y J, et al. Multi-moduli NTTs for saber on cortex-m3 and cortex-m4. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, (1): 127-151

[33] Paksoy I K, Cenk M. Faster NTRU on ARM cortex-m4 with TMVP-based multiplication. IEEE Transactions on Circuits and Systems I: Regular Papers, 2022, 69(10): 4083-4092

[34] Fan H, Hasan M A. A new approach to subquadratic space complexity parallel multipliers for extended binary fields. IEEE Transactions on Computers, 2007, 56(2): 224-233

[35] Alkim E, Cheng D Y L, Chung C M M, et al. Polynomial multiplication in NTRU prime: Comparison of optimization strategies on cortex-m4. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, (1): 217-238

[36] Good I J. Random motiyon a finite Abelian group. Mathematical Proceedings of the Cambridge Philosophical Society, 1951, 47(4): 756-762

[37] Huang J, Zhang J, Zhao H, et al. Improved plantard arithmetic for lattice-based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, (4): 614-636

[38] Plantard T. Efficient word size modular arithmetic. IEEE Transactions on Emerging Topics in Computing, 2021, 9(3): 1506-1518

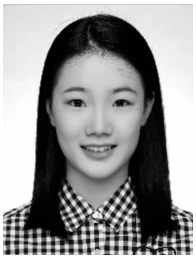
[39] Cooley J W, Tukey J W. An algorithm for the machine calculation of complex fourier series. Mathematics of Computation, 1965, 19(90): 297-301

[40] Gentleman W M, Sande G. Fast Fourier transforms: For fun and profit//Proceedings of the AFIPS'66 Fall Joint Computer Conference. San Francisco, USA, 1966: 563-578

[41] Güneysu T, Oder T, Pöppelmann T, et al. Software speed records for lattice-based signatures//Proceedings of the Post-Quantum Cryptography: The 5th International Workshop (PQCrypto 2013). Limoges, France, 2013: 67-82

[42] Alkim E, Jakubeit P, Schwabe P. Newhope on ARM Cortex-M//Proceedings of the International Conference on Security, Privacy, and Applied Cryptography Engineering. Hyderabad, India, 2016: 332-349

[43] Duman J, Hövelmanns K, Kiltz E, et al. Faster lattice-based KEMs via a generic fujisaki-okamoto transform using prefix hashing//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Virtual Event, Republic of Korea, 2021: 2722-2737



WEI Han-Yu, M. S. candidate. Her main research interests include post-quantum cryptography and cryptographic engineering.

ZHENG Jie-Yu, Ph. D. candidate. Her main research interests include post-quantum cryptography and cryptographic engineering.

ZHAO Yun-Lei, Ph. D., professor. His main research interests include post-quantum cryptography, cryptographic protocols, and theory of computing.

Background

This paper focuses on the compact and efficient implementation of the lattice-based CNTR/CTRU KEM based on ARM Cortex-M4 platform. This problem belongs to software optimization in cryptographic engineering. Deploying PQC algorithms on ARM Cortex-M4 will provide more reliable guarantees for the security of IoT devices. The embedded platform implementation of the lattice-based cryptography scheme is very important for prompting the progress of PQC algorithm standard. Although the CNTR/CTRU schemes have C and AVX2 optimization, ARM Cortex-M4 implementation of these two schemes is still blank. This paper presents a high-speed and low-stack ARM Cortex-M4 design of CNTR/CTRU schemes. The main contributions include: (1) For the first time, polynomial CBD sampling is implemented on ARM Cortex-M4, speeding up the process of sampling. (2) Using mixed-radix NTT to speed up NTT-unfriendly polynomial multiplication, making full use of FPU, using layer merging in NTT to minimize time-consuming operations such as load and store. (3) Delaying reduction by NTT process coefficient range analysis, thereby decreasing the number of reductions, and decreasing the number of assembly instructions for Barrett reduction and Montgomery reduction. (4) Using loop unrolling to achieve polynomial inversion, optimizing the time-consuming process of polynomial inversion. (5) the combination of multi-moduli NTT and CRT is used to accelerate the speed of the decryption process, which includes NTT-unfriendly modulus polynomial ring multiplication. (6) Using space multiplexing to optimize the stack usage. The previous research directions of our research group include software and hardware optimization of Kyber, Dilithium and so on.

This research was partially supported by the National Key Research and Development Program of China (No. 2022-YFB2701600), the General Project of State Key Laboratory of Cryptography (No. MMKFKT202227), the Technical Standard Project of Shanghai Scientific and Technological Committee (No. 21DZ2200500), the Shanghai Collaborative Innovation Fund (No. XTCX-KJ-2023-54), and the Special Fund for Key Technologies in Blockchain of Shanghai Scientific and Technological Committee (No. 23511100300).