中图分类号：TN918.4　　论文编号：1028716 24-B033
学科分类号：083900

博士学位论文

公钥密码算法优化实现研究

研究生姓名　　张吉鹏
学科、专业　　网络空间安全
研 究 方 向　　密码工程
指 导 教 师　　Çetin Kaya Koç 教授

南京航空航天大学
研究生院　计算机科学与技术学院

二〇二五年二月

Nanjing University of Aeronautics and Astronautics
The Graduate School
College of Computer Science and Technology

Research on Optimized Implementation of Public-key Cryptography Algorithms

A Thesis in

Cyberspace Security

by

Jipeng Zhang

Advised by

Prof. Çetin Kaya Koç

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

February, 2025

承诺书

本人声明所呈交的博士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：_____

日　　期：_____

摘　要

公钥密码算法在现行的网络安全系统中占据着重要地位,本文主要研究椭圆曲线密码(ECC)和后量子密码（PQC）这两类公钥密码算法。相比于 RSA 密码体制，ECC 在保持高安全性的同时，具有较低的计算复杂度和更短的密钥长度，这使得 ECC 逐步成为 TLS、SSH 和 IPSec 等网络协议中应用最为广泛的密码算法之一。后量子密码技术则在量子计算威胁日益加剧的背景下，为未来的网络空间安全提供了有效的的解决方案。椭圆曲线密码与后量子密码两者的结合，有望在未来的网络空间安全中继续发挥重要作用，为数据安全保驾护航。

对于 ECC，本文的研究涵盖了我国商密标准 SM2 以及国际标准 X/Ed25519，所面临的困难与挑战如下：首先，尽管现有的密码算法库如 OpenSSL 和 GmSSL 支持 SM2，但由于未对 ARMv8-A 架构进行专门适配和优化，导致其在国产 ARM 处理器上的性能欠佳。其次，ECC 的核心运算不具备直观的并行计算流程，难以充分利用处理器的 SIMD 并行计算能力。此外，优化后的密码实现难以集成到复杂的 TLS 协议中。对于 PQC，本文的研究对象是基于模格的 Saber 方案和 NIST 所制定的基于模格的 ML-KEM 和 ML-DSA 标准，研究动机如下：多项式乘法在格密码中占据核心耗时运算，并依赖复杂的 NTT 算法。嵌入式平台的内存资源限制使得基于模格的密码算法部署困难。最后，格密码中伪随机数生成依赖的 SHA-3 算法计算复杂，性能优化难度大。在上述问题的驱动下，本文从以下四个方面对 ECC 和 PQC 的优化实现开展了研究：

1. 本文研究了我国商用密码标准 SM2 数字签名算法在国产华为 ARM 鲲鹏处理器上的优化实现。虽然著名的 OpenSSL 与 GmSSL 代码库提供了 SM2 的支持，但其在国产 ARM 处理器上的性能仍有较大的提升空间。对于有限域运算，本文针对 SM2 模数的数值特征提出了优化的蒙哥马利模乘 CIOS 方法，针对模逆运算推导并实现了更快的基于费马小定理的模逆算法，本文优化的模乘与模逆实现是 OpenSSL 实现性能的 1.3 倍和 2.1 倍。对于固定点和非固定点标量乘运算，本文分别实现了宽度为 7 和 5 的窗口算法，分别是 OpenSSL 实现性能的 30.4 倍和 5.2 倍。对于签名生成的计算，本文提出了更快的计算方法，使用一个模加/减运算替换一个模乘运算从而加速签名生成的执行。本文还将优化的 SM2 实现集成到了 OpenSSL 的性能测试框架中，在华为云鲲鹏 920 上的测试表明，签名与验签性能分别是 OpenSSL 实现的 8.7 倍和 3.5 倍。
2. 本文探讨了国际通用 ECC 标准 X/Ed25519 算法在 Intel AVX-512 上的并行性优化策略及其向 TLS 应用的集成。ECC 因缺乏直观的并行执行流因此面临着并行实现难的问题，并且优化实现难以向复杂的 TLS 软件栈集成。对于有限域运算，本文设计并实现了 8 路并

行策略以最大化 AVX-512 的并行计算能力，并使用形式化验证工具 CryptoLine 对有限域实现进行了验证从而保证其正确性和鲁棒性。对于椭圆曲线点运算和标量乘运算，本文充分探索了各种并行实现技术，以充分发挥 8 路有限域运算的并行能力。性能测试表明，本文的 X/Ed25519 实现性能至高可达 OpenSSL 实现的 12 倍。除此以外，本文还研究了如何在不修改 TLS 协议和 TLS 应用代码的情况下，将优化的 X/Ed25519 实现集成至 TLS 应用中，还探索了如何缓解 AVX2/AVX-512 执行单元的冷启动问题。最终设计了三种端到端实验来对本文的解决方案进行测试，测试表明 DNS over TLS 服务端的峰值吞吐至多可提升 41%。

3. 本文研究了双发射 RISC-V 处理器上 NTT 多项式乘法的优化方法。Saber 方案以及 NIST 所制定的后量子密码标准 ML-KEM 与 ML-DSA 均基于模格进行构造。本文研究了如何在双发射 RV32IM、双发射 RV64IM 以及 RISC-V Vector 处理器上优化实现 ML-KEM 与 ML-DSA 的 NTT 多项式乘法，对于有限域模乘运算，本文探索了如何使用改进版 Plantard 模乘算法对 ML-KEM 与 ML-DSA 的 NTT 进行优化实现，并给出了在双发射处理器上的流水线优化方法，实现性能至多可达之前实现的 29.9 倍。对于基于模格的 Saber 方案，其原本的参数设定并不支持 NTT，本文则研究了如何为其适配 NTT 算法，本文的实现性能提升至多可达 55%；本文还提出了针对 Saber 方案的多种内存优化技术，使得 Saber 方案的整体内存占用低于 5KB，有助于促进其在资源受限的物联网设备上的部署和应用。
4. 本文研究了格密码的另一个耗时运算 SHA-3 的性能优化。SHA-3 的核心组件 Keccak-f1600 由于计算复杂且 1600 比特的内部状态增大了寄存器分配难度，因此优化实现难度较大。本工作在双发射 RISC-V 处理器上的 Keccak-f1600 优化实现涵盖了各种常见的指令集组合，包括：RV32IM、RV32IMB、RV64IM、RV64IMB 和 RVV，并且还论证了标量-向量混合实现的可行性。本工作重新审视了各种优化实现技术，并为各种指令集组合上的实现构造了最优技术组合，在平头哥玄铁 C908 处理器上的测试表明，本工作的各种实现的性能提升至高可达 98%。本工作还将优化的 NTT 多项式乘法和 Keccak-f1600 实现集成到了 ML-KEM 与 ML-DSA 方案中，测试表明本工作在各种 RISC-V 指令集组合上的实现均刷新了性能记录，性能分别至多可达之前实现的 4.7 倍和 4.2 倍，在一定程度上丰富了 SHA-3 以及后量子密码在 RISC-V 上的软件生态。

关键词：密码工程，椭圆曲线密码优化实现，格密码优化实现，后量子密码优化实现

ABSTRACT

Public key cryptographic algorithms play a crucial role in current cybersecurity systems. This thesis focuses on the study of two types of public key cryptographic algorithms: Elliptic Curve Cryptography (ECC) and Post-Quantum Cryptography (PQC). Compared to the RSA encryption system, ECC maintains high security while offering lower computational complexity and shorter key lengths. This makes ECC one of the most widely used cryptographic algorithms in network protocols like TLS, SSH, and IPSec. On the other hand, PQC provides an effective solution for future cybersecurity in the face of growing quantum computing threats. The combination of ECC and PQC is expected to continue playing a significant role in ensuring data security in the evolving landscape of network security.

The research on ECC in this thesis focuses on China's national commercial cryptographic standard SM2 and the international X/Ed25519 standards. The challenges and difficulties faced in this area include the following: First, although existing cryptographic libraries such as OpenSSL and GmSSL support SM2, they are not specifically optimized for the ARMv8-A architecture, resulting in suboptimal performance on domestic ARM processors. Second, ECC's core operations lack an intuitive parallel execution flow, making it difficult to fully leverage the SIMD parallelism of processors. Furthermore, optimized cryptographic implementations are hard to integrate into complex protocols like TLS. In the field of PQC, this thesis examines the lattice-based Saber scheme and the lattice-based ML-KEM and ML-DSA standards defined by NIST. The motivation behind this research includes the following: Polynomial multiplication is a core time-consuming operation in lattice-based cryptography and relies on complex NTT algorithms. The limited memory resources of embedded platforms make the deployment of lattice-based cryptographic algorithms challenging. Additionally, the performance optimization of the SHA-3 algorithm, which is crucial for generating pseudorandom numbers in lattice cryptography, presents significant difficulties. To address these challenges, this thesis explores optimization strategies for both ECC and PQC in four key areas:

1. Optimization of the SM2 digital signature algorithm on Huawei's ARM Kunpeng processors: While widely-used libraries like OpenSSL and GmSSL support SM2, significant performance improvements can be made on domestic ARM processors. For finite field operations, this thesis proposes an optimized Montgomery modular multiplication CIOS method tailored to the numerical characteristics of the SM2 modulus and develops a faster modular inverse algorithm based on Fermat's Little Theorem. These optimizations achieve performance improvements of 1.3x and 2.1x

over the OpenSSL implementation for modular multiplication and modular inversion, respectively. Furthermore, optimized scalar multiplication methods based on window algorithms achieve performance improvements of 30.4x and 5.2x, respectively, compared to OpenSSL. When integrated into OpenSSL's performance testing framework, the optimized SM2 implementation shows up to 8.7x and 3.5x improvements in signing and verification performance on the Huawei Kunpeng 920 processor.

2. Parallel optimization of the X/Ed25519 algorithm on Intel AVX-512: ECC algorithms face challenges in parallel execution due to their lack of an intuitive parallel execution flow. This thesis designs and implements an 8-way parallel strategy for finite field operations to fully utilize AVX-512's parallel capabilities, and employs formal verification tools like CryptoLine to ensure the correctness and robustness of the implementation. For elliptic curve point operations and scalar multiplication, various parallelization techniques are explored to maximize the parallelism of finite field operations. Performance tests demonstrate up to a 12x improvement over OpenSSL implementations. Additionally, the thesis explores integrating the optimized X/Ed25519 implementation into TLS applications without modifying the TLS protocol or application code and addresses the cold-start issues of AVX2/AVX-512 execution units. End-to-end tests show up to a 41% improvement in peak throughput for DNS over TLS servers.
3. Optimization of NTT polynomial multiplication on dual-issue RISC-V processors: This thesis investigates how to optimize the NTT polynomial multiplication in the Saber scheme and NIST's ML-KEM and ML-DSA standards on dual-issue RV32IM, RV64IM, and RISC-V Vector processors. For finite field modular multiplication, the thesis explores optimizing NTT operations using an improved Plantard modular multiplication algorithm and introduces pipeline optimization methods for dual-issue processors, resulting in up to a 29.9x performance improvement. Additionally, the thesis adapts the NTT algorithm for the Saber scheme, resulting in up to a 55% performance improvement. Memory optimization techniques for the Saber scheme are also proposed, reducing overall memory usage to less than 5KB, facilitating deployment in resource-constrained IoT devices.
4. Performance optimization of SHA-3 in lattice-based cryptography: SHA-3, particularly its core component Keccak-f1600, is a time-consuming operation in lattice-based cryptography due to its complexity and the challenges of managing the large 1600-bit internal state. This thesis presents optimized implementations of Keccak-f1600 on dual-issue RISC-V processors, exploring various instruction set combinations, including RV32IM, RV32IMB, RV64IM, RV64IMB, and RVV. The feasibility of scalar-vector hybrid implementations is also examined. Performance tests on

the XuanTie C908 processor show performance improvements of up to 98%. Furthermore, the optimized NTT polynomial multiplication and Keccak-f1600 implementations are integrated into the ML-KEM and ML-DSA schemes, yielding up to 4.7x and 4.2x performance improvements, respectively. These results contribute to enhancing the software ecosystem for SHA-3 and post-quantum cryptography on RISC-V platforms.

Keywords: Cryptographic engineering, Optimized implementation of elliptic curve cryptography, Optimized implementation of lattice-based cryptography, Optimized implementation of post-quantum cryptography

目　　录

图表清单

算法清单

代码片段清单

注释表

记号	描述	记号	描述
R_q	$R_q = \mathbb{Z}_q[X]/(X^n+1)$	n_{sm2}	SM2 椭圆曲线加法群基点阶
a_0, a_1, a_2, a_3	多项式系数	$F_{p_{25519}}$	Curve25519 底层有限域
G, P, Q	椭圆曲线点	p_{25519}	Curve25519 模数
(x, y)	仿射坐标点	n_{25519}	Curve25519 基点阶
(X, Y, Z)	雅可比投影坐标点	SHA512	哈希函数
(X, Y, Z, T)	扩展坐标点	ζ	本原根
∞	椭圆曲线无穷远点	$\hat{f}$	多项式 f 的 NTT 域表示
$E(F_p)$	椭圆曲线加法群	$\mathsf{ek}_{\mathsf{PKE}}$	加密密钥
kG	椭圆曲线标量乘	$\mathsf{dk}_{\mathsf{PKE}}$	解密密钥
$F_{p_{sm2}}$	SM2 底层有限域	ek	封装密钥
p_{sm2}	SM2 模数	dk	解封装密钥
pk	公钥	sk	私钥

缩略词

缩略词	英文全称
ECC	Elliptic Curve Cryptography
DH	Diffie-Hellman
ECDH	Elliptic Curve Diffie-Hellman
DSA	Digital Signature Algorithm
ECDLP	Elliptic Curve Discrete Logarithm Problem
PQC	Post-Quantum Cryptography
CACR	Chinese Association for Cryptologic Research
IoT	Internet of Things
SIMD	Single Instruction Multiple Data
AVX2	Advanced Vector Extensions 2
AVX-512	Advanced Vector Extensions 512
RAM	Random Access Memory
ROM	Read Only Memory
AEAD	Authenticated Encryption with Associated Data
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HMAC	Hash-based Message Authentication Code
TLS	Transport Layer Security
TCP	Transmission Control Protocol
IETF	Internet Engineering Task Force
SHA-3	Secure Hash Algorithm 3
ISA	Instruction Set Architecture
CPI	Cycles Per Instruction
IPC	Instructions Per Cycle
RAW	Read After Write
NTT	Number Theoretic Transform
RVV	RISC-V Vector

第一章　绪论

公钥密码算法在如今的网络空间安全中发挥着不可或缺的作用，本文主要研究椭圆曲线密码和后量子密码算法这两类公钥密码算法的优化实现。对于椭圆曲线密码算法，本文的研究涵盖我国商密 SM2 数字签名算法与国际通用标准 X/Ed25519 算法；对于后量子密码，本文主要针对格密码进行研究。本文所涉及的处理器包括以物联网为代表的嵌入式处理器和以服务器为代表的高性能处理器。本章首先介绍研究背景，然后介绍公钥密码算法的国内外研究现状，然后介绍本文的研究动机与研究内容，最后介绍本文的组织结构。

1.1　研究背景

椭圆曲线密码（Elliptic Curve Cryptography, ECC）[1,2] 相比于 RSA 算法具有诸多优点：（1）在相同的安全强度下，基于 ECC 的密码协议，比如椭圆曲线迪菲-赫尔曼密钥交换协议（ECDH），比 RSA 和迪菲-赫尔曼密钥交换协议的运算速度快。（2）在相同的安全强度下，ECC 的密钥和曲线参数比 RSA 小很多。（3）ECC 体制的困难性基于椭圆曲线离散对数问题（ECDLP）的难解性，目前计算 ECDLP 的通用算法都是指数级的困难度。考虑到上述优势，ECC 已逐步取代 RSA 成为 TLS[3,4]、SSH[5–8]、IPSec[9] 等网络协议中应用最为广泛的公钥密码算法。

2020 年 12 月 3 日，《科学》杂志在线发表了中国科学技术大学潘建伟教授研究团队与中科院上海微系统所、国家并行计算机工程技术研究中心合作的研究成果[10]——“九章量子计算机”。“九章量子计算机” 是 76 个光子 100 个模式的量子计算原型机，成功地实现了 “高斯玻色取样” 任务的快速求解。根据现有理论，该系统处理高斯玻色取样的速度比目前最快的超级计算机快一百万亿倍。未来，随着更大规模的量子计算机的研制成功，传统的公钥密码体系将面临巨大的安全威胁，特别是依赖于 RSA 和 ECC 等难题的密码系统将不再安全。因此，后量子密码（Post-Quantum Cryptography, PQC）逐渐成为密码学界的研究重点。后量子密码不仅能够抵御量子计算机的攻击，还能防范传统计算机的攻击。包括格密码、编码密码和同源密码在内的多种密码体制正在广泛研究中，其中格密码因其优异的安全性和性能成为最受欢迎的一类。

美国国家标准与技术研究院（NIST）于 2016 年开始征集后量子密码候选标准，经过多轮评估后，于 2024 年 8 月正式发布 FIPS 203[11]、FIPS 204[12] 和 FIPS 205[13] 文件，分别基于 Kyber[14]、Dilithium[15] 和 SPHINCS$^+$[16] 标准化了 ML-KEM、ML-DSA 和 SLH-DSA 算法。中国密码学会（CACR）也在 2018 年 6 月主办了全国密码算法设计竞赛，竞赛向外界征集并评估分组密码和公钥密码，其中公钥密码的提交算法均为后量子密码。该竞赛一共进行了两轮，2020 年 1 月初公布了最终评选结果，其中 Aigis 和 LAC 算法获得公钥密码竞赛一等奖。

表 1.1　各类嵌入式处理器的具体规格

处理器	架构	RAM	ROM	频率
AT90CAN64	8 位 AVR	4 KB	64 KB	16 MHz
ATxmega192A3	8 位 AVR	16 KB	192 KB	32 MHz
ATSAM3X8E	32 位 ARM Cortex-M3	100 KB	512 KB	84 MHz
STM32F407VGT6	32 位 ARM Cortex-M4	192 KB	1 MB	168 MHz
SiFive Freedom E310	32 位 RISC-V	16 KB	4 MB	320 MHz

随着物联网（IoT）、无线传感网络（WSNs）和移动自组织网络（MANETs）等领域的快速发展，嵌入式芯片在传感器和移动设备中扮演着越来越重要的角色。在智能家居、智能交通、智能电网、智慧旅游、工业自动化以及车辆通信等领域，物联网的应用正日益增多。统计数据显示，2022 年全球已部署了数百亿个传感器设备和移动设备。由于这些设备常被部署在高风险环境中，易受到窃听和恶意消息注入等安全威胁，部署如 ECC 和 PQC 等公钥密码系统以保证系统、数据和事务安全变得至关重要。

个人电脑与高性能服务器所使用的 CPU 包括 Intel/AMD 的 x86-64 系列和 ARM 的 A 系列 CPU（如基于 ARMv8-A/ARMv9-A 架构的苹果 M 系列 SoC）。这些高性能 CPU 支持 SIMD 指令，其中 Intel/AMD 的 CPU 支持 256 位宽的 AVX2 指令，部分还支持 512 位宽的 AVX-512 指令；ARMv8-A/ARMv9-A 系列 CPU 则支持 128 位宽的 NEON 指令。这些 SIMD 指令不仅用于加速多媒体应用，还可用于加速密码运算，特别是公钥密码运算。

与个人电脑和高性能服务器不同，嵌入式设备多由电池供电，通常需要长时间运行。这些设备在电池容量、计算能力和存储资源上都非常有限。例如，AVR 和 MSP 芯片的运行频率一般低于 20 MHz，而 ARM Cortex-M 系列芯片最多也只有 200 MHz。在存储资源方面，AVR 和 MSP 芯片的 RAM 容量通常小于 10 KB，ROM 约为 100 KB 左右；而 ARM Cortex-M 系列芯片则较为丰富，如常见的 ARM Cortex-M4 和 ARM Cortex-M3 系列芯片通常有大于 100 KB 的 RAM，表 1.1 汇总了各类嵌入式芯片的具体规格。

1.2　国内外研究现状

1.2.1　椭圆曲线密码发展历史与优化实现现状

发展历史　自 ECC 的概念被提出以来，ECC 经历了显著的发展和广泛的应用。下文将梳理 ECC 的发展历史、各种类型的椭圆曲线以及相关的标准。下文将 ECC 的发展划分为两个阶段：

（1）ECC 的理论发展阶段。1985 年，Neal Koblitz 和 Victor Miller 分别独立提出了基于椭圆曲线的密码学方法[1,2]。他们的工作展示了椭圆曲线在公钥密码学中的潜力，特别是在密钥交换

和数字签名方面。与传统的 RSA 和离散对数系统相比，ECC 在提供相同安全级别的情况下，所需的密钥长度更短，从而提高了效率。在 ECC 提出后的几年里，研究人员对其数学基础和安全性进行了深入研究。Gura 等人的研究[17] 表明，即使在 8 比特 AVR 嵌入式处理器上，ECC 的计算性能也能优于 RSA。ECC 的早期应用主要集中在学术界，逐步扩展到实际应用中。

（2）ECC 的标准化与应用阶段。美国国家标准与技术研究院（NIST）在 2000 年发布了 FIPS 186-2 标准[18]，推荐了一系列基于椭圆曲线的密码算法，这些标准曲线包括 P-192、P-224、P-256、P-384 和 P-521，覆盖了不同的安全级别和应用需求。互联网工程任务组（IETF）也在多个协议中采用了 ECC。例如，RFC 4492 定义了在 TLS 中使用 ECC 的扩展，RFC 7748 则定义了 Curve25519 和 Curve448 用于密钥交换[19]，这两个算法的安全级别分别为 128 比特和 224 比特，得益于巧妙的模数选取，其计算性能普遍优于其他算法。这些标准推动了 ECC 在互联网安全中的广泛应用。除了 NIST 和 IETF，其他标准组织如 ISO、ANSI 和 SECG 也制定了基于 ECC 的标准。这些标准涵盖了密钥交换、数字签名和加密等多个方面，促进了 ECC 在全球范围内的应用。ECC 除了应用在 TLS 协议以外，在不经意传输（Oblivious Transfer，OT）协议中也有着广泛的应用，文献[20] 便使用 X25519 算法实现了多种 OT 协议。

优化实现现状 多精度蒙哥马利模乘是一些 ECC 算法中的关键运算，诸如 NIST P-256 与我国商用密码 SM2 算法的实现均依赖于多精度蒙哥马利模乘算法。1996 年，Koc 等人[21] 分析了五种多精度蒙哥马利模乘实现方法，分别是 SOS、CIOS、FIOS、FIPS 和 CIHS。他们的实验表明将多精度乘法和蒙哥马利约减相结合执行的 CIOS 算法性能最优，这得益于 CIOS 算法在乘法指令和加法指令数量保持不变的情况下，减少了内存访问。

ECC 在 ARM Cortex-A 系列处理器上的研究可追溯至 2012 年，Bernstein 等人[22] 在 ARM Cortex-A8 处理器上，使用 NEON 指令优化了 Curve25519 算法。该工作主要优化实现了模乘运算，他们使用有符号数 $2^{25.5}$ 的冗余表示方法来表示有限域上的大整数，并充分利用模数 $2^{255}-19$ 的特征来加速模约减运算，最终性能测试表明计算共享密钥仅需 527102 个时钟周期，而同时期 OpenSSL 中的 NIST P-256 曲线则需要近 9 百万时钟周期。

Intel/AMD 的 x86-64 系列处理器在个人电脑与计算中心等业务场景具有广泛的应用，因此使用 x86-64 指令集优化 ECC 受到了学界广泛关注。Mai 等人[23] 在 x86-64 处理器上优化了 SM2 数字签名算法，其优化包含四个方面：在大整数运算中充分利用了 `mulx`、`adcx` 和 `adox` 指令对有限域运算进行优化；通过将大整数乘法和蒙哥马利约减算法结合起来执行来降低临时寄存器的使用；针对固定点标量乘法，他们预计算了 256 个椭圆曲线点并使用窗口算法进行实现；利用 wNAF 方法实现了非固定点标量乘法。最终在 Haswell 微架构上的性能测试表明，SM2 签名与验签的吞吐分别为每秒 97475 次和 18870 次。Gueron 等人[24] 在 x86-64 处理器上对 NIST P-256 进行了充分优化，他们利用模数的数值特征优化了蒙哥马利模乘算法。针对固定点标量

乘法，他们使用宽度 $w = 7$ 的窗口算法并结合预计算技术进行优化实现。针对非固定点标量乘法，则使用了 $w = 5$ 的窗口算法。在 Haswell 微架构处理器上，相比于同时期的 OpenSSL 实现，性能提升 1.8 倍 ~ 2.3 倍。

如何使用 SIMD 指令优化 ECC 是密码工程领域内一个值得研究的课题，当代处理器常见的 SIMD 指令集包括 Intel/AMD x86-64 处理器中的 AVX2 和 AVX-512，以及 ARMv8-A 架构中的 NEON 指令集。Faz-Hernández 等人[25] 使用 AVX2 指令优化了 Curve25519 和 Curve448，他们的优化与文献[22] 的思路类似，利用了模数 $2^{255} - 19$ 的数值特性，并结合 AVX2 的并行性来加速 Curve25519 和 Curve448。最终在 Haswell 微架构的处理器上的测试表明，相比于之前的最快实现，Ed25519 性能提升 15% ~ 24%，Ed448 性能提升 21% ~ 22%，X25519 性能提升 11%，X448 性能提升 20%。2020 年，Cheng 等人[26] 利用 AVX2 指令集以高吞吐为优化目标实现了 4×1 路的 X25519 实现，他们的底层有限域、椭圆曲线点加/倍点以及上层的密钥生成和共享密钥计算均采用 4×1 路并行策略，从而能获得较高的吞吐量。最终在 Haswell 微架构的处理器上的测试表明，密钥生成与共享密钥计算的吞吐分别提升 60.4% 和 45.7%。但多路实现的密码算法如何集成至 TLS 应用中，并是否能使 TLS 受益仍需要进一步研究。

1.2.2 后量子密码发展历史与优化实现现状

发展历史 后量子密码（PQC）的概念最早可以追溯到 20 世纪 90 年代，下文将简单梳理后量子密码的发展历史：

（1）PQC 的早期发展阶段。1994 年，彼得 · 肖尔（Peter Shor）提出了一种有效的量子算法，可以在多项式时间内因数分解大整数，这对 RSA 和 ECC 等传统公钥密码体系构成了直接威胁。随之而来，学者们意识到需要开发新型的密码方案，以抵御未来量子计算的攻击。在 2000 年代初期，研究人员开始探索各种能够抵抗量子计算攻击的密码学方案。主要的研究方向包括格密码、基于编码的密码、基于多变量的密码、基于哈希的签名方案等。

（2）PQC 的标准化阶段。为了促进 PQC 的广泛应用，NIST 于 2016 年启动了后量子密码标准化项目，旨在评估和标准化多种 PQC 方案。经过多轮评估，NIST 于 2024 年 8 月正式发布了 FIPS 203、FIPS 204 和 FIPS 205 文件，分别标准化了 ML-KEM、ML-DSA 和 SLH-DSA 算法。

（3）PQC 的部署与应用。2022 年发布的 OpenSSH 9.0[27] 中便部署了 NTRU Prime 方案，其采用 NTRU Prime 与 X25519 混合的方式来进行密钥交换。2024 年 8 月，瑞典 VPN 技术提供商 Mullvad 宣布[28]，其能提供后量子保护的 VPN 应用已实现支持 iPhone 系统，并已在 Linux、Windows、MacOS、Android 和 iPhone 等主流平台上部署了后量子与传统加密的混合算法。这些应用基于 WireGuard 协议，并支持 McEliece 和 ML-KEM 算法。2024 年 9 月，微软公司宣布[29] 其开源核心加密库 SymCrypt 已支持后量子加密算法，以帮助保护其客户免受未来的量子威胁。

优化实现现状 格密码中的核心运算是多项式乘法，多项式乘法常使用 NTT（Number Theoretic Transform）算法进行加速，其中的关键运算是整数模乘运算。Seiler 提出了一种有符号数蒙哥马利模乘变体[30]，在此之前，格密码的 SIMD 实现中不得不使用无符号数版本的蒙哥马利模乘算法或使用浮点数进行实现，无符号数版本蒙哥马利模乘算法的缺点是计算过程会产生两倍宽度的中间值，因此会导致并行性减半；对于使用浮点数实现的方法[31]，使用 32 位单精度浮点数会损失计算精度，因此必须使用 64 位双精度浮点数来表示一个多项式整数系数，而对于 Kyber 和 NewHope 方案，其模数小于 2^{16}，因此浮点数实现方法也损失了并行性。Seiler 提出的有符号数蒙哥马利模乘变体计算过程中不会产生两倍宽度的中间值，因此在 AVX2 指令集上的实现显著优于其他实现方法。最终在 Skylake 微架构上的测试表明，Seiler 的方法能分别使 NewHope 和 Kyber 的多项式乘法性能提升 4.2 倍和 6.3 倍。这一项工作成为后续各种格密码方案的 NTT 多项式乘法在各种处理器上优化实现的基础。

对于另一常用的模乘方法，Barrett 算法，也有相关工作对其进行优化和改进。Becker 等人提出了一种适用于 NEON 指令集的 Barrett 模乘算法，并提供了其在 ARMv8-A NEON 指令集上的实现[32]。NEON 上的模乘实现与 AVX2 上的实现略有不同，AVX2 的乘法指令支持只获取高或低半部分结果，因此能高效地实现蒙哥马利模乘算法，而 NEON 的乘法指令只能得到双倍宽度的乘法结果，因此会使并行性减半。该工作提出的 Barrett 模乘则能充分地利用 NEON 中的 `sqrdmulh` 指令，不会导致并行性减半，因此优于蒙哥马利模乘实现。最终的实验表明，在 ARM Cortex-A72 处理器上，Kyber 方案比之前的实现快 9% ~ 13%，Saber 方案比之前的实现快 31% ~ 35%。

格密码在嵌入式平台上的研究备受学界关注，常用的嵌入式平台包括 ARM Cortex-M3、ARM Cortex-M4 以及 32 位 RISC-V 处理器。Botros 等人针对 Kyber 在 ARM Cortex-M4 平台上进行了优化实现[33]，其 NTT 实现性能是之前最快实现的 2 倍，并且大幅度降低了 Kyber 的内存占用。Botros 等人的具体优化方法包括：（1）由于 `pqm4` 代码库将不同功能的子程序实现放到了不同的文件中，并分别编译这些源文件，这使得编译器无法内联实现这些子程序来进行性能优化。该工作通过实验表明，编译时加入`-flto` 选项即可缓解这一问题，并可使 Kyber 的整体性能提升 5%。（2）对于 NTT 的性能优化：采用有符号表示法；使用 2 层合并而不是之前工作所采用的 3 层合并策略；使用有符号数蒙哥马利模乘算法；将 2 个 16 位多项式系数 “打包” 放入一个 32 位寄存器中，并使用 Cortex-M4 所提供的 SIMD 指令集来实现双路并行计算。（3）对于内存优化：考虑到多项式矩阵 **A** 在计算过程中只被用到一次，因此采用即时生成策略，即一次只生成一个所需的多项式，因此能降低内存占用。

Greconici 等人在 ARM Cortex-M3 平台上对 Dilithium 进行了性能和内存优化[34]。对于 ARM Cortex-M3 上的实现，因为其 32 位乘法指令是非恒定执行时间的，因此该工作使用 16 位乘法指令来实现 32 位乘法操作，从而保证 NTT 实现是恒定执行时间的。对于内存优化，该工作设计了

3 种不同的时间-内存权衡策略，使得 Dilithium 签名验证的内存消耗仅为 10KB 左右。Alkim 等人在 ARM Cortex-M4 平台上对 NewHope 和 Kyber 进行了优化，并给出了首个 NewHope-Compact 在 Cortex-M4 上的实现[35]。该工作的优化方法包括：（1）对于有符号数蒙哥马利约减实现，通过预计算 $-q^{-1}$ 并使用 `smlabb` 指令，蒙哥马利约减只需使用 2 条指令，相比于之前的实现节省了一条指令。（2）使用`-flto` 编译选项来进行链接时优化。（3）对于内存优化，主要的思路是使用即时生成策略来计算多项式乘法从而降低内存占用。最终的性能测试表明，该工作的优化能使 Kyber 和 NewHope 性能提升 10% 左右。Greconici 给出了 Kyber 在 32 位 RISC-V 处理器上的优化实现[36]，得益于 RISC-V 架构所提供的足够寄存器，NTT 实现中最多可完成 4 层合并策略，该工作还使用汇编语言优化了蒙哥马利模乘和 Barrett 约减算法，最终在 PQVexRiscV 上的测试表明，相比于参考实现，性能提升为 70% ~ 83%。

上述相关工作所研究的 NewHope、Kyber、Dilithium 等密码方案的多项式环支持 NTT 多项式乘法，而诸如 Saber、LAC、NTRU、NTRU Prime 等密码方案的多项式环则并不能直接支持 NTT 算法，这类多项式环称为 NTT 不友好的多项式环。Chung 等人为 Saber 和 NTRU 这两个 NTT 不友好的多项式环方案在 ARM Cortex-M4 和 AVX2 上适配了 NTT 多项式乘法[37]。对于 Saber 方案，在 ARM Cortex-M4 上，该工作的实现相比于之前实现 CPU 时钟周期减少了 21% ~ 38%，在 AVX2 上，CPU 时钟周期减少了 2% ~ 30%。对于 NTRU 方案，在 ARM Cortex-M4 上，该工作的实现相比于之前实现 CPU 时钟周期减少了 3% ~ 13%；在 AVX2 上，CPU 时钟周期减少了 7% ~ 15%。除此以外，本工作还为 LAC 方案适配了 NTT 多项式乘法，使得 LAC 在 ARM Cortex-M4 上的性能提升 10 倍以上，在 AVX2 上性能提升 3 ~ 7 倍。

除了利用各种处理器的标准指令集或 SIMD 指令集的实现外，Albrecht 等人研究了如何利用 RSA 协处理器来加速 NewHope 和 Kyber 方案[38]，该工作利用 Kronecker 置换算法将多项式乘法转换为大整数乘法，从而利用 RSA 协处理器所提供的大整数计算能力来加速多项式运算。简言之，该工作通过利用 Kronecker 算法以及相关变体能将 Kyber-768 解封装中的多项式乘法运算转换为 120 个 2049 比特模乘，而 2048 比特安全性的 RSA 则需要接近 3072 个 1024 比特乘法。

1.3 研究动机与研究内容

本文的主要研究对象是椭圆曲线密码和后量子密码这两类公钥密码算法，研究椭圆曲线密码是因为其目前在工业界具有广泛的应用场景，研究后量子密码则是为未来后量子迁移场景做技术储备。对于后量子密码迁移，目前学术界与工业界普遍认可的方案是将传统公钥密码算法（如 ECC）与后量子密码混合使用[39]，这也侧面印证了本文同时研究 ECC 与 PQC 的意义。对于椭圆曲线密码算法的研究，本文的研究涵盖了我国商用密码标准 SM2 以及国际通用标准 X/Ed25519；对于后量子密码，本文的研究对象是 NIST 所制定的基于模格的后量子密码标准

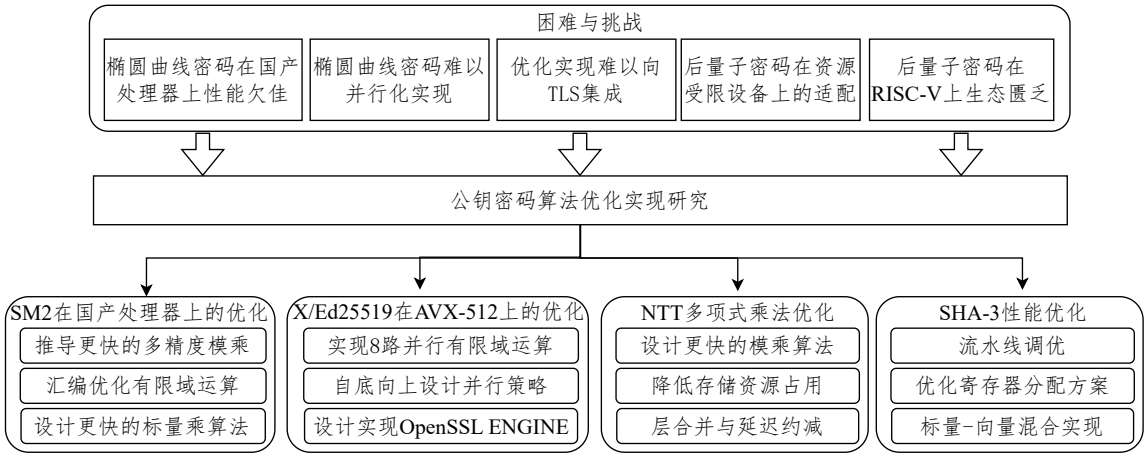


图 1.1　本文研究工作总体思路图

算法 ML-KEM 与 ML-DSA，并主要针对其中的两个耗时运算展开研究，即 NTT 多项式乘法与 SHA-3。图 1.1 给出了本文研究工作总体思路图。

本文针对椭圆曲线密码的研究动机如下：

- SM2 在国产处理器上性能欠佳：虽然目前主流的密码算法库 OpenSSL 和 GmSSL 提供了对商密 SM2 的支持，但并没有针对 ARMv8-A 架构进行专门的适配和优化，导致其在国产 ARM 处理器上性能欠佳。
- ECC 难以并行化实现：ECC 的核心运算从底向上分别为有限域运算、椭圆曲线点运算和标量乘运算，这些运算均不具备直观的并行计算流程，因此难以受益于处理器的 SIMD 并行计算能力。
- 优化实现向 TLS 集成难度大：以著名的 OpenSSL 密码库为例，其工程复杂性较高，因此使得优化的密码实现难以向复杂的 TLS 软件栈集成。
- AVX 冷启动问题严重影响性能：Intel AVX2、AVX-512 执行单元的冷启动问题会导致密码算法遭受严重的性能降级，甚至会使某些原语受到 3.8 倍的性能降级。

在上述因素的启发下，本文展开如下两点的研究，分别对应本文第三章和第四章：

- 商密 SM2 在国产华为鲲鹏处理器上的优化实现研究：本文对 SM2 进行了自下而上的优化，在有限域层推导更快的多精度模乘并使用 ARMv8-A 汇编指令充分地优化有限域运算；针对标量乘运算，本文基于窗口算法设计并实现了比 OpenSSL 实现更快的标量乘算法。
- 国际 ECC 标准 X/Ed25519 算法在高性能 Intel 服务器上的优化实现研究：为了更好地利用 AVX-512 指令集的并行性，本文充分研究了各个层的并行实现策略，并给出了目前已知的最优并行优化策略；针对向 TLS 的集成，本文基于 OpenSSL ENGINE API 设计并实现了 ENG25519，其能将优化的 X/Ed25519 实现集成至 TLS 应用，进而探索密码优化对

应用层的改进。本工作还提出了基于启发式的“热身”线程来缓解冷启动问题。

本文针对后量子密码的研究动机如下：

- 多项式乘法在格密码中耗时占比高：格密码的核心耗时运算之一是多项式乘法，并常用时间复杂度为 $O(nlogn)$ 的 NTT 算法实现，其中的核心运算是整数模乘。
- 嵌入式平台内存资源受限使得格密码部署困难：正如表 1.1 所示，部分嵌入式平台的 RAM 规格只有 16KB 甚至更低，而基于模格的密码算法因为计算涉及到多项式矩阵，需占用较多的内存资源，使得其在资源受限的平台上部署较为困难。
- SHA-3 是格密码的核心耗时运算且计算复杂：格密码中伪随机数的生成依赖于 SHA-3 中的 SHAKE 算法，其也是格密码中的耗时运算之一。SHA-3 的核心组件是 Keccak-f1600 算法，其计算涉及到 1600 比特的状态，并且计算逻辑颇为复杂，使得其性能优化难度较大。

在上述因素的启发下，本文展开如下两点的研究，分别对应本文第五章和第六章：

- 适用于格密码的高效模乘算法与 NTT 优化实现研究：本文研究了改进版 Plantard 模乘算法在双发射 RV32IM 与 RV64IM 上的优化实现方法及其在 ML-KEM 与 ML-DSA 方案中的应用，研究了 RISC-V Vector 指令集上 NTT 多项式乘法的优化实现，还研究了如何为 Saber 方案适配 NTT 多项式乘法以及如何优化 Saber 方案的内存占用从而使其能部署到资源实现的物联网设备上。
- SHA-3 性能优化研究及其在格密码中的应用：本文详细回顾了 Keccak-f1600 算法的各种优化实现技术，并针对各种嵌入式处理器构造了最优技术组合，这些研究进一步丰富了 SHA-3 以及格密码在各种嵌入式处理器上的软件生态。

研究内容的关联性与整体性 本文针对公钥密码中的两类密码算法进行研究，分别是椭圆曲线密码与后量子密码，下面通过回答如下问题来讲解本文研究内容的关联性和整体性：

- 本文四个工作的整体性：第三四章研究椭圆曲线密码是因为其是目前主流应用场景中使用最广泛的一类算法，对其进行研究能使目前主流应用受益。随着量子计算威胁的加剧，第五六章研究后量子密码算法是为互联网基础设施向抗量子时代迁移做技术储备。简言之，对椭圆曲线密码和后量子密码算法的研究分别面向当前应用场景与未来抗量子迁移场景。除此以外，在向后量子密码标准迁移的过程中，学术界与工业界普遍认可的方法是将椭圆曲线密码与后量子密码混合使用[39]，这也印证了本文同时研究 ECC 与 PQC 的价值和意义。这两类密码算法相辅相成，在未来后量子密码迁移过程中将一起发挥重要作用。
- 椭圆曲线密码算法以及实现平台是如何选择的，即第三章与第四章的关系是什么？本文对椭圆曲线密码算法的研究中，具体算法的选择既包含我国商用密码标准 SM2 也包含国

际通用标准 X/Ed25519，实现平台的选择同时兼顾了国产服务器平台与国际通用 x86-64 平台，因此既支撑了国产算法与国产硬件的生态构建，也兼顾了国际通用标准的支持。

- 后量子密码的具体优化方法是什么，即第五章与第六章的关系是什么？本文对后量子密码算法的研究中，重点研究了格密码的两个关键耗时组件，即多项式乘法与 SHA-3，分别对应本文第五章与第六章，并且本文的研究兼顾内存优化与性能优化。既刷新了格密码在各个平台上的性能记录，也进一步丰富了格密码以及 SHA-3 的软件生态。除此以外，本文的研究也能对我国国产后量子密码的性能优化产生一定的启发意义。

1.4 组织结构

本文主要围绕椭圆曲线密码与后量子密码这两类公钥密码算法的性能优化展开研究，对于椭圆曲线密码算法，本文的研究涵盖我国商密 SM2 与国际通用标准 X/Ed25519；对于后量子密码，本文的研究对象是基于模格的 Saber 方案和 NIST 所制定的基于模格的 ML-KEM 与 ML-DSA 标准，主要研究其中的多项式乘法与 SHA-3 这两个核心运算。本文的章节组织如下：

第二章主要介绍了符号定义、相关基础知识，主要对有限域运算和环上的多项式乘法进行介绍。本章还介绍了本工作所涉及的几个密码方案：商密 SM2 数字签名算法、X/Ed25519 算法、基于模格的 Saber 方案以及 NIST 制定的后量子密码标准 ML-KEM 与 ML-DSA 方案。

第三章研究了商密 SM2 在国产华为鲲鹏处理器上的优化实现。具体来说，本章针对 SM2 的有限域运算提出了优化的蒙哥马利模乘 CIOS 方法，并推导实现了更快的基于费马小定理的模逆运算。对于固定点与非固定点标量乘运算，本章使用窗口算法并结合预计算技术进行优化实现。本章还提出了签名计算过程中签名项 s 的更快的计算方法。最终，本章将上述优化实现技术集成到了 OpenSSL 的性能测试框架中，并在华为云鲲鹏 920 处理器上进行了性能测试，SM2 签名与验签性能分别是之前实现的 8.7 倍与 3.5 倍。该工作有望进一步推进 SM2 在国产 ARM 服务器上的应用和普及。

第四章研究了国际通用 ECC 标准 X/Ed25519 算法在高性能 Intel 服务器上利用 AVX-512 指令集的优化实现。具体来说，本章设计并实现了 8×1 路有限域并行策略，并证明 8 路策略可避免置换指令因此相比于其他策略能提升运行效率，还使用 CryptoLine 工具对本章的有限域实现进行了形式化验证以保证正确性和鲁棒性。基于 8×1 路有限域并行策略，本章设计并实现了 8×1 路和 2×4 路椭圆曲线点运算策略，这两种策略具有不同的用途，前者能充分发挥 8×1 路有限域运算的并行能力，后者则可用于构造 1×2 路双点标量乘运算。对于标量乘运算，本章也设计并实现了多种并行策略，包括 8×1 路、1×8 路和 1×2 路策略，不同的策略用于不同的原语。上述优化实现在 Intel Xeon (Ice Lake) Platinum 8369B 处理器上的测试表明，本章的实现相比于之前已知的最快实现加速比例最高可达 12.01 倍。除了上述密码算法的优化实现，本章还研究了如何在不修改 TLS 协议和 TLS 应用代码的前提下，将优化的密码实现集成至 TLS 协议

与 TLS 应用中，并进一步研究了如何缓解 AVX2/AVX-512 执行单元的冷启动问题。最终构造了三种不同的端到端实验来验证本章优化实现的效果，最高可使 DNS over TLS 服务端的峰值吞吐提高 41%。该研究能使大规模互联网服务提供商在提升用户体验、增强服务可用性以及降低运营成本等方面受益。

第五章研究了格密码中 NTT 多项式乘法的优化实现，具体来说，本章研究了如何为 Saber 方案适配 NTT 算法，以及如何在双发射 RV32IM、双发射 RV64IM 和 RVV 处理器上优化 ML-KEM 与 ML-DSA 方案的 NTT 多项式乘法。Saber 方案原本的参数设定并不支持 NTT 多项式乘法，本章研究了如何为 Saber 适配 NTT 算法并研究了 Saber 方案的内存优化方法，本章的实现不仅刷新了性能记录，而且还将 Saber 的内存占用降低至 5KB 以内，有助于推动 Saber 方案在内存受限的物联网设备上的部署与应用。本章指出，ML-KEM 方案的 NTT 在 RV32IM 和 RV64IM 上可使用改进版 Plantard 模乘，ML-DSA 方案的 NTT 只能在 RV64IM 上使用改进版 Plantard 模乘。本章重点研究了 NTT 多项式乘法在双发射处理器上的流水线优化方法，提出的流水线调优技术能减少因乘法指令耗时较高所导致的流水线停顿。最终的性能测试表明，本章的实现均刷新了性能记录，以 ML-KEM 方案的 NTT 多项式乘法在玄铁 C908 处理器上的实现为例，本章的优化实现性能最高可达之前实现的 29.9 倍。该研究能有效促进格密码在各种嵌入式处理器上的应用和普及，并对我国国产自主可控的后量子密码在嵌入式平台上的高效实现与实际部署具有一定的启发作用。

第六章研究了格密码的另一个耗时运算 SHA-3 的性能优化。SHA-3 的核心组件是 Keccak-f1600 算法，对于 Keccak-f1600 在 RISC-V 处理器上的优化，本章的研究涵盖了各种常见的指令集组合，并重新审视了各种优化实现技术。对于 RV64I 上的实现，本章设计了专门的流水线优化方案来改进 Keccak-f1600 的性能；对于 RV64IB 上的实现，本章阐述了如何利用 B 扩展提供的 `rori` 和 `andn` 指令来加速 Keccak-f1600；对于 RV32I 和 RV32IB 上的实现，针对寄存器资源紧张的难题，本章以流水线友好性为主要目标，设计了专门的寄存器分配方案和双发射流水线优化方法；对于 RVV 上的实现，本章不仅阐述了如何利用向量指令实现 Keccak-f1600，还探索了标量-向量混合实现技术，论证了混合实现技术在各种指令集组合上的可行性。最终在玄铁 C908 处理器上的性能测试表明，本章 Keccak-f1600 实现性能最多是之前实现性能的 4 倍。与此同时，本章还将优化的 Keccak-f1600 实现集成到了 ML-KEM 与 ML-DSA 方案中，在各种处理器平台上均刷新了性能记录。该工作不仅能促进格密码在各种嵌入式处理器上的应用和普及，还能进一步丰富后量子密码和 SHA-3 的软件生态。

第七章进行了全文总结，对本文的贡献进行总结，并展望了未来工作的思路。

第二章　预备知识

本章主要给出一些常用符号及说明、椭圆曲线密码和后量子密码的相关背景与数学知识。对于椭圆曲线密码的研究，本文主要针对国密 SM2 数字签名算法与国际 ECC 标准 X/Ed25519 算法，因此在 2.2 节中给出 SM2 的介绍，在 2.3 节中给出 X25519 与 Ed25519 的介绍。对于后量子密码的研究，本文主要针对 NIST 所制定的后量子密码标准 ML-KEM 与 ML-DSA 方案以及基于 MLWR 问题的 Saber 方案，因此在 2.4 节中给出 ML-KEM 的介绍，在 2.5 节中给出 ML-DSA 的介绍，在 2.6 节中给出 Saber 的介绍。

2.1　符号定义

在描述椭圆曲线密码时，小写字母如椭圆曲线系数 $a_1 \sim a_6$、椭圆曲线点坐标 (x,y) 和标量 k 均为有限域 F_p 内的元素，p 为有限域运算的模数，n 为 ECC 基点的阶。小写字母 $\tilde{a}$ 和 $\tilde{b}$ 表示元素 a 和 b 在蒙哥马利域中所对应的元素。大写字母如 G、P 和 Q 表示椭圆曲线上的点，椭圆曲线点在仿射（Affine）坐标中可表示为 (x, y)，在雅可比投影（Jacobian Projective）坐标中用 (X, Y, Z) 表示，在扩展坐标中用 (X, Y, Z, T) 表示。椭圆曲线所有点的集合用 E/F_p 表示。大写字母 W 表示机器字长，小写字母 w 表示计算标量乘法的窗口算法窗口宽度。

在描述基于模格的后量子密码时，小写字母 n 通常表示模格的维度或者模格的大小，k 与 ℓ 通常表示模格中的子系统数量或者模格的子格数量，q 通常表示有限域模数；形如 $\eta_1, \eta_2, d_u, d_v, \tau$ 则是一些安全参数。R_q 用于表示多项式环 $\mathbb{Z}_q[X]/(X^n+1)$，其中的多项式表示为 $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$，其中 $f_j \in \mathbb{Z}_q$，其中的整数运算需模 q，多项式运算需模 X^n+1。NTT 常用于表示正向 NTT 变换，INTT 用于表示 NTT 的逆。对于 $\hat{f} := NTT(f)$，其中 $\hat{f}$ 表示多项式 f 的 NTT 域表示。

2.2　商密 SM2 数字签名算法

2010 年国家密码管理局公布了一系列商用密码算法标准，其中包括 SM2 椭圆曲线公钥密码算法标准[40]，并提供了 128 比特安全性的推荐参数，参数选取具有高度自主可控性。2018 年，SM2 数字签名算法成功入选 ISO/IEC 标准[41]，由此可见 SM2 数字签名算法的实现效率与安全性得到了国内外学界的认可，同时著名的 OpenSSL 与 GmSSL 密码库也提供了 SM2 数字签名算法的实现。SM2 包含三种公钥密码协议：数字签名、密钥交换和公钥加密，本文主要针对最为常用的数字签名进行研究。

ECC 的运算自底向上可以分成四个层次：有限域运算、椭圆曲线加法群上点的运算、标量

乘法运算以及协议层运算，下文分别对 SM2 的这四个层次展开介绍。SM2 底层有限域为 $F_{p_{sm2}}$，其中 $p_{sm2} = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$。因为 p_{sm2} 的位数大于机器字长，在计算机中需使用多个机器字来表示，所以常称有限域 $F_{p_{sm2}}$ 中的元素为大整数。

定义在 $F_{p_{sm2}}$ 上的 Weierstrass 等式表示为 $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$，其中系数 $a_1 \sim a_6 \in F_{p_{sm2}}$。SM2 与 NIST P-256 类似，采用了简化后的短 Weierstrass 曲线：

$$y^2 = x^3 + ax + b \mod p_{sm2} \tag{2.1}$$

其中 $a, b \in F_{p_{sm2}}$, $\Delta = 4a^3 + 27b^2 \neq 0$，且令 $a = -3$ 以获得更快的椭圆曲线点运算。

对于有限域 F_p，所有满足某椭圆曲线等式的点 (x, y) 与一个无穷远点 ∞ 作为单位元共同构成一个椭圆曲线加法群，常记为 $E(F_p)$，该群中的点加和倍点运算按照弦切定理计算。P 和 G 为椭圆曲线点，$k \in F_p$，标量乘法 $P = kG$ 的基本运算单元为椭圆曲线加法群中的点加和倍点运算。椭圆曲线加法群中存在一个基点 G，其它所有椭圆曲线点均可以通过与基点的标量乘法计算得到。基点的阶为 n，使得 $nG = \infty$ 成立。

2.2.1 有限域运算

有限域 F_p 上的核心运算包括模加/减、模乘/平方和模逆运算。模加/减运算较为简单，但涉及条件加/减 p 的运算，而条件加/减法如果使用分支语句，如 C 语言中的 if-else 语句，则会产生存在安全风险的非恒定运行时间的 ECC 实现，因此通常采用掩码技术实现条件加/减运算以得到恒定运行时间的 ECC 实现。模乘/平方运算是有限域中较为耗时且使用频繁的运算，这两种运算的优化对 ECC 整体运行效率的提升至关重要，因此下文重点介绍模乘/平方运算的实现细节。

模乘/平方运算的实现可分为两个步骤：(1)乘法/平方运算，通常使用教科书乘法（schoolbook multiplication）进行实现，乘数或被乘数的位数大于机器字长时称为大整数或多精度乘法/平方运算；(2) 模约减运算，常用蒙哥马利约减算法或基于模数数值特征的快速约减算法进行实现。相比于借助非恒定时间除法运算实现的模约减方法（$c = c - \lfloor c/p \rfloor \cdot p \mod p$），蒙哥马利约减与快速约减都是恒定时间的。这两种算法的基本思路为：

- 快速约减算法：该算法利用模数的等价关系将两个大整数乘积中权重高于 2^{256} 的字转换成与低权重字的加/减运算；对于 SM2，等价关系为 $2^{256} = 2^{224} + 2^{96} - 2^{64} + 1 \mod p_{sm2}$。快速约减算法只适用于具有良好数值特性的模数，SM2 中的模 n_{sm2} 约减运算则无法使用此算法，n_{sm2} 为椭圆曲线加法群中基点的阶。相较于具有更简单结构的 Curve25519 模数 $p_{25519} = 2^{255} - 19$，针对 SM2 模数 p_{sm2} 的快速约减算法更复杂。
- 蒙哥马利约减算法：该算法的基本思想是使用移位运算替换非恒定执行时间的除法运算。

算法 2.1: 蒙哥马利模乘 CIOS 方法

输入: 有限域 F_p 中的元素 $a = \sum_{i=0}^{3} a_i 2^{64i}$, $b = \sum_{i=0}^{3} b_i 2^{64i}$; 模数 p; 常量 $p' = -p^{-1} \mod 2^{256}$; 常量 $\beta = 2^{256}$

输出: $t = ab\beta^{-1} \mod p$

1 **for** $i \leftarrow 0$ *to* 3 **do**
2 $C \leftarrow 0$
3 **for** $j \leftarrow 0$ *to* 3 **do**
4 $(C, S) \leftarrow t_j + a_j b_i + C$
5 $t_j \leftarrow S$
6 **end**
7 $(C, S) \leftarrow t_4 + C$
8 $t_4 \leftarrow S, t_5 \leftarrow C$; /* $t = b_i a + t$ */
9 $m \leftarrow t_0 p'_0 \mod 2^{64}$; /* $p'_0 = p' \mod 2^{64}$ */
10 $(C, S) \leftarrow t_0 + m p_0$
11 **for** $j \leftarrow 1$ *to* 3 **do**
12 $(C, S) \leftarrow t_j + m p_j + C$
13 $t_{j-1} \leftarrow S$
14 **end**
15 $(C, S) \leftarrow t_4 + C$
16 $t_3 \leftarrow S$
17 $t_4 \leftarrow t_5 + C$
18 **end**
19 **if** $t \geq p$ **then** $t \leftarrow t - = p$
20 **return** t

令 $\beta = 2^{Ws}$，其中 W 为机器字长，s 为大整数字数，且要求 $\beta > p$，$\mathrm{GCD}(\beta, p) = 1$。假设大整数 $a, b \in F_p$ 的乘积为 t。蒙哥马利约减算法计算 $c = t\beta^{-1} \mod p$，其计算原理为 $c = (t + (tp' \mod \beta)\, p)\, /\beta$，其中 $p' = -p^{-1} \mod \beta$，除以 β 与右移 Ws 位等价。最后还需要执行一次条件减 p 运算。

实现模乘/平方运算时，通常将多精度乘法与模约减结合起来执行，如蒙哥马利模乘 CIOS 方法[21]，其 64 位实现如算法 2.1 所示。该算法将蒙哥马利约减嵌入到多精度乘法的外循环中，每次计算完 $t \leftarrow b_i a + t$，便对乘积 t 进行约减。算法 2.1 的约减部分每次循环对低 64 位字进行约减，如算法 2.1 第 10 行省略了 $t_0 + m p_0$ 的低半部分乘积 S。以此类推，外层循环每次约减乘积的一个 64 位字，循环结束后即可完成对乘积 t 的约减运算。最后需要对 t 进行条件减 p 使结果处于 $[0, p)$ 范围内。

此外，有限域中最为复杂的运算为模逆运算。模逆运算可以通过扩展欧几里德算法（辗转相

除法）实现，但其运行时间通常与输入相关，是非恒定时间的。本章采用费马小定理（$a^{-1} = a^{p-2} \bmod p$）来实现恒定时间的模逆运算。

2.2.2 椭圆曲线点运算

椭圆曲线点运算主要包括点加与倍点运算，其计算效率与椭圆曲线点的表示方法有着重要的关系。下面对 SM2 数字签名算法所依赖的短 Weierstrass 曲线上的点加/倍点运算进行介绍。

在仿射坐标下，椭圆曲线点表示为 (x, y)。仿射坐标下短 Weierstrass 曲线上点的取负运算公式为 $-(x, y) = (x, -y)$。根据弦切定理，仿射坐标下短 Weierstrass 曲线上点加运算 $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ 与倍点运算 $(x_3, y_3) = 2(x_1, y_1)$ 的计算公式分别为：

$$\begin{cases} x_3 = ((y_2 - y_1)/(x_2 - x_1))^2 - x_1 - x_2, \\ y_3 = (y_2 - y_1)(x_1 - x_3)/(x_2 - x_1) - y_1 \end{cases} \tag{2.2}$$

$$\begin{cases} x_3 = \left((3x_1^2 + a)/2y_1\right)^2 - 2x_1 \\ y_3 = (3x_1^2 + a)(x_1 - x_3)/2y_1 - y_1 \end{cases} \tag{2.3}$$

其中 a 为式 2.1 中的常量。

在仿射坐标下计算点加/倍点涉及复杂的模逆运算，因此通常利用坐标转换来减少模逆运算。以下文即将介绍的用于加速短 Weierstrass 曲线上的点加/倍点运算的雅可比投影坐标为例，雅可比投影坐标上的点加/倍点运算无需计算模逆操作，因此常常先在雅可比投影坐标上执行一系列的点加/倍点运算，最终再将结果转换到仿射坐标上，从而能减少模逆运算的次数。

对于 SM2 数字签名算法所依赖的短 Weierstrass 曲线，通常使用三坐标的雅可比投影坐标 $(X, Y, Z), Z \neq 0$ 来表示椭圆曲线点。雅可比投影坐标点 (X, Y, Z) 与仿射坐标点 $(X/Z^2, Y/Z^3)$ 等价。雅可比投影坐标下短 Weierstrass 曲线上点的取负运算公式为 $-(X, Y, Z) = (X, -Y, Z)$。将 $x = X/Z^2, y = Y/Z^3$ 分别代入式 2.2 和式 2.3 即可得到雅可比投影坐标下的点加运算 $(X_3, Y_3, Z_3) = (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2)$ 和倍点运算 $(X_3, Y_3, Z_3) = 2(X_1, Y_1, Z_1)$ 计算公式[42] 分别为：

$$\begin{cases} X_3 = \left(Y_2 Z_1^3 - Y_1\right)^2 - \left(X_2 Z_1^2 - X_1\right)^2 \left(X_1 + X_2 Z_1^2\right) \\ Y_3 = \left(Y_2 Z_1^3 - Y_1\right) \left(X_1 \left(X_2 Z_1^2 - X_1\right)^2 - X_3\right) - Y_1 \left(X_2 Z_1^2 - X_1\right)^3 \\ Z_3 = \left(X_2 Z_1^2 - X_1\right) Z_1 \end{cases} \tag{2.4}$$

$$\begin{cases} X_3 = \left(3X_1^2 + aZ_1^4\right)^2 - 8X_1 Y_1^2 \\ Y_3 = \left(3X_1^2 + aZ_1^4\right) \left(4X_1 Y_1^2 - X_3\right) - 8Y_1^4 \\ Z_3 = 2Y_1 Z_1 \end{cases} \tag{2.5}$$

算法 2.2: 从左到右二进制扫描的标量乘算法

输入: 标量 k，其二进制表示为 $(k_{l-1}, \cdots, k_1, k_0)_2$，$l$ 为 k 的比特长度；P 为一椭圆曲线点

输出: 标量乘法结果 $Q = kP$

```
Q ← ∞
for i ← l − 1 downto 0 do
    Q ← 2Q
    if k_i ← 1 then
        Q ← Q + P
    end
end
return Q
```

算法 2.3: 基于窗口的标量乘算法，窗口宽度为 w

输入: 标量 $k = \sum_{i=0}^{\lceil l/w \rceil - 1} k_i 2^{wi}$；$P$ 是一椭圆曲线点

输出: 标量乘法结果 $Q = kP$

```
预计算
  table[i] ← iP, i = 0, ··· , 2^w − 1。
  i = 0 时 table[0] ← ∞
Q ← ∞
for i ← ⌈l/w⌉ − 1 downto 0 do
    Q ← 2^w Q
    Q ← Q + table[k_i]
end
return Q
```

其中 a 为式 2.1 中的常量。

2.2.3 标量乘法运算

标量乘法运算是 ECC 的核心运算，根据所涉及的椭圆曲线点是否已知，可将标量乘法分为固定点标量乘和非固定点标量乘。算法 2.2 所描述的从左到右二进制扫描算法是标量乘最简单且通用的实现方法，其基本思路为：逐比特从左向右扫描标量 k，每次循环先执行一次倍点运算，然后按照 k_i 的值执行一次条件点加运算，以此类推直到扫描到 k 的最低比特 k_0，其中 l 为标量的比特长度。该算法存在条件点加运算，容易遭受侧信道攻击。因此为了保证安全性，需要实现具有固定执行流程和固定时间的标量乘法运算。除此以外，该算法每次只扫描标量的一个比特，执行效率较低。

算法 2.3 所描述的窗口算法比上述二进制扫描算法更为高效。该算法先将一定数量的椭圆曲线点预计算并保存在预计算表中，每次扫描标量的 w 个比特片段，根据标量片段的值从预计算表中取椭圆曲线点并进行运算。对于固定点标量乘法，预计算可在代码运行前执行因此不消耗执行时间; 而非固定点标量乘法的预计算则需要在运行时执行。以固定点标量乘为例，假设标量 k 的二进制表示中比特 1 与比特 0 各占一半，算法 2.2 大约需要 $l/2$ 次点加运算，而算法 2.3 的点加次数为 $\lceil l/w \rceil$。该算法是以空间换时间的策略，可以有效提升标量乘法运算的运行效率。

对于 SM2 数字签名算法，签名生成过程中涉及到一次固定点标量乘，即 kG，G 为已知的基点；签名验证过程中涉及到一次固定点标量乘和一次非固定点标量乘，即 $s'G + tP_A$，G 为已知的基点，P_A 为未知点，该运算也常称为双点标量乘。

更进一步，观察到 ECC 点的减法运算可以通过点的取负运算和点加运算实现，因此点的加

算法 2.4: SM2 数字签名生成算法

输入: 签名者 A 的杂凑值 Z_A；待签名消息 M；签名者 A 的公私钥对 (d_A, P_A)

输出: 数字签名对 (r, s)

$e \leftarrow H_v(Z_A||M)$

$r \leftarrow 0;\ s \leftarrow 0$

while $s = 0$ **do**

  while $r = 0$ 或 $r + k = n_{sm2}$ **do**

    Random$(k) \in [1, n_{sm2} - 1]$; /* 生成随机数 */

    $(x_1, y_1) \leftarrow kG$

    $r \leftarrow e + x_1 \mod n_{sm2}$

  end

  $s \leftarrow ((1 + d_A)^{-1} \cdot (k - rd_A)) \mod n_{sm2}$

end

return (r, s)

法和减法计算开销相差不大，这个特性可用于减少预计算表大小。比如，在不改变标量 k 值的前提下，将标量 k 的每 w 比特都转换成有符号数表示，如使用 NAF 编码[42]。利用有符号数表示，算法 2.3 的预计算点数从 $2^w - 1$ 减少到了 $2^{w-1} - 1$。窗口宽度 w 的选择涉及时间与空间，即计算开销与预计算表存储空间之间的权衡，需结合具体情况进行分析。

2.2.4 协议层运算

算法 2.4 为 SM2 数字签名生成算法，该算法的输入包括用户 A 的杂凑值 Z_A、待签名消息 M 和用户 A 的公私钥对。首先利用哈希函数 H_v 计算 Z_A 与 M 的 v 比特哈希值 e。然后随机选取 $k \in [1, n_{sm2} - 1]$ 并计算标量乘 $(x_1, y_1) = kG$ 与 $r = e + x_1 \mod n_{sm2}$。然后再利用 r、k 和用户 A 的私钥 d_A 计算另一个签名项 s。最后，如果 $r = 0$ 或 $r + k = n_{sm2}$，则需要重新选择 k 并重新计算数字签名对，否则直接输出签名对 (r, s)。

数字签名验证算法如算法 2.5 所示，该算法由用户 B 执行，输入包括用户 A 的杂凑值 Z_A、A 的公钥 P_A、待验证的签名消息 M' 以及签名对 (r', s')。在 r' 与 s' 符合正常取值范围的情况下，先计算 Z_A 和 M' 的哈希值与 $t = r' + s' \mod n_{sm2}$。如果 $t \neq 0$ 则计算两个标量乘法之和 $(x_1', y_1') = s'G + tP_A$，最后判断 $r = e + x_1' \mod n_{sm2}$ 是否与待验证的签名项 r' 相等。当 $r = r'$ 时，签名验证正确，否则签名验证失败。

2.3 X25519 与 Ed25519 算法

以蒙哥马利（Montgomery）曲线[43] 为基础的 X25519 密钥协商算法[44] 和以扭曲 Edwards 曲线[45] 为基础的 Ed25519 数字签名算法[46] 由于其公开的设计以及优异的性能，逐渐成为目

算法 2.5: SM2 数字签名验证算法

输入: 签名者 A 的杂凑值 Z_A；签名者 A 的公钥 P_A；待验证消息 M'；待验证签名对 (r', s')

输出: 签名验证结果：成功或失败

if $r' \notin [1, n_{sm2} - 1]$ 或 $s' \notin [1, n_{sm2} - 1]$ **then**
  return 失败
end
$e' \leftarrow H_v(Z_A||M')$
$t \leftarrow r' + s' \mod n_{sm2}$
if $t = 0$ **then**
  return 失败
end
$(x'_1, y'_1) \leftarrow s'G + tP_A$
$r \leftarrow e + x'_1 \mod n_{sm2}$
if $r = r'$ **then**
  return 成功
else
  return 失败
end

前主流的国际 ECC 标准，并在很多网络协议中被广泛使用，包括但不限于 TLS[3,4]、SSH[5–8]、IPSec[9]、OpenPGP[47]、DNSCurve[48]、X3DH[49]、Double Ratchet[50] 和 MLS[51] 等协议。Curve25519 是 X/Ed25519 底层所依赖的椭圆曲线[52]。

Curve25519 底层有限域为 $F_{p_{25519}}$，其中 $p_{25519} = 2^{255} - 19$。因为 p_{25519} 的位数大于机器字长，在计算机中需使用多个机器字来表示，所以常称有限域 $F_{p_{25519}}$ 中的元素为大整数。

X25519 密钥协商算法所依赖的 Curve25519 曲线的蒙哥马利形式表示为：

$$y^2 = x^3 + ax^2 + x \mod p_{25519} \tag{2.6}$$

其中 $a = 486662$。

Ed25519 数字签名算法所依赖的 Curve25519 曲线的扭曲 Edwards 形式与所对应的蒙哥马利形式是双向等价的，表示为：

$$-x^2 + y^2 = 1 - dx^2y^2 \mod p_{25519} \tag{2.7}$$

其中 $d = 121665/121666$。

2.3.1 有限域运算

本文 2.2.1 中给出了有限域运算的基础介绍，因此不再赘述。值得一提的是，在实现 $F_{p_{25519}}$ 上的模乘/模平方时常采用快速约减算法，该算法利用模数的等价关系将两个大整数乘积中权重高于 2^{256} 的字转换成低权重字的加/减运算，对于 Curve25519，等价关系为 $2^{255} = 19 \mod p_{25519}$。

2.3.2 椭圆曲线点运算

下面对 X25519 密钥协商算法所依赖的蒙哥马利曲线和 Ed25519 数字签名算法所依赖的扭曲 Edwards 曲线上的点加/倍点运算进行介绍。

仿射坐标下蒙哥马利曲线上点的取负运算公式为 $-(x, y) = (x, -y)$。仿射坐标下蒙哥马利曲线上点加运算 $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ 与倍点运算 $(x_3, y_3) = 2(x_1, y_1)$ 的计算公式分别为：

$$\begin{cases} x_3 = ((y_2 - y_1)/(x_2 - x_1))^2 - a - x_1 - x_2, \\ y_3 = (y_2 - y_1)(2x_1 + x_2 + a)/(x_2 - x_1) - ((y_2 - y_1)/(x_2 - x_1))^3 - y_1 \end{cases} \tag{2.8}$$

$$\begin{cases} x_3 = \left((3x_1^2 + 2ax_1 + 1)/2y_1\right)^2 - a - 2x_1 \\ y_3 = (3x_1^2 + 2ax_1 + 1)(3x_1 + a)/2y_1 - ((3x_1^2 + 2ax_1 + 1)/2y_1)^3 - y_1 \end{cases} \tag{2.9}$$

其中 a 为式 2.6 中的常量。

仿射坐标下扭曲 Edwards 曲线上点的取负运算公式为 $-(x, y) = (-x, y)$。仿射坐标下扭曲 Edwards 曲线上点加运算 $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ 与倍点运算 $(x_3, y_3) = 2(x_1, y_1)$ 的计算公式分别为：

$$\begin{cases} x_3 = (x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2), \\ y_3 = (y_1y_2 + x_1x_2)/(1 - dx_1x_2y_1y_2) \end{cases} \tag{2.10}$$

$$\begin{cases} x_3 = 2x_1y_1/(1 + dx_1^2y_1^2), \\ y_3 = (y_1^2 + x_1^2)/(1 - dx_1^2y_1^2) \end{cases} \tag{2.11}$$

其中 d 为式 2.7 中的常量。

对于 X25519 密钥协商算法所依赖的蒙哥马利曲线，Montgomery[43] 发现可以不使用仿射坐标中的 y 坐标即可完成点加/倍点运算，因此使用 (X, Z) 坐标进行运算，其中 $x = X/Z$，且常使用差分点加公式和倍点公式直接构造蒙哥马利阶梯算法来计算标量乘法，蒙哥马利阶梯算法将在下文进行介绍。差分点加是指点加计算需要提前得知两点的差值，这一前提条件在蒙哥马利阶梯算法中可以得到满足。已知椭圆曲线点 $P = (X_2, Z_2)$、$Q = (X_3, Z_3)$ 并已知两点的差值

$P-Q=(X_1,Z_1)$，同时计算倍点 $(X_4,Z_4)=2(X_2,Z_2)$ 和点加 $(X_5,Z_5)=P+Q$ 的公式为：

$$\begin{cases} A=X_2+Z_2 \\ B=X_2-Z_2 \\ C=X_3+Z_3 \\ D=X_3-Z_3 \\ X_4=A^2B^2 \\ Z_4=(A^2-B^2)(B^2+((a+2)/4)(A^2-B^2)) \\ X_5=Z_1(AD+BC)^2 \\ Z_5=X_1(AD-BC)^2 \end{cases} \tag{2.12}$$

其中 A,B,C,D 为临时变量，a 为式 2.6 中的常量，常量 $(a+2)/4$ 可以被预计算，两点差值的坐标即 (X_1,Z_1) 只在计算点加时用得到，将倍点与点加计算公式写在一起是因为这两个公式在蒙哥马利阶梯算法中常被同时计算。

对于 Ed25519 数字签名算法所依赖的扭曲 Edwards 曲线，常使用扩展坐标[45](X,Y,Z,T) 进行计算，其与仿射坐标 (x,y) 的关系为：$XY=ZT$，$x=X/Z$，$y=Y/Z$。扩展坐标下扭曲 Edwards 曲线上点的取负运算公式为 $-(X,Y,T,Z)=(-X,Y,-T,Z)$。将上述关系分别带入式 2.10 和式 2.11 即可推导出扩展坐标下的点加 $(X_3,Y_3,T_3,Z_3)=(X_1,Y_1,T_1,Z_1)+(X_2,Y_2,T_2,Z_2)$ 和倍点 $(X_3,Y_3,T_3,Z_3)=2(X_1,Y_1,T_1,Z_1)$ 运算公式分别为：

$$\begin{cases} X_3=(X_1Y_2+Y_1X_2)(Z_1Z_2-dT_1T_2) \\ Y_3=(Y_1Y_2+X_1X_2)(Z_1Z_2+dT_1T_2) \\ T_3=(Y_1Y_2+X_1X_2)(X_1Y_2+Y_1X_2) \\ Z_3=(Z_1Z_2-dT_1T_2)(Z_1Z_2+dT_1T_2) \end{cases} \tag{2.13}$$

$$\begin{cases} X_3=2X_1Y_1(2Z_1^2-Y_1^2+X_1^2) \\ Y_3=(Y_1^2-X_1^2)(Y_1^2+X_1^2) \\ T_3=2X_1Y_1(Y_1^2+X_1^2) \\ Z_3=(Y_1^2-X_1^2)(2Z_1^2-Y_1^2+X_1^2) \end{cases} \tag{2.14}$$

其中 d 为式 2.7 中的常量。

2.3.3 标量乘法运算

对于 X25519 密钥协商算法，在根据本端私钥和对端公钥计算共享密钥（shared secret）时所涉及的非固定点标量乘常使用蒙哥马利梯子算法进行实现，见算法 2.6。算法 2.6 采用上文所

算法 2.6: 蒙哥马利阶梯算法

输入: 标量 k，其二进制表示为 $(k_{l-1}, \cdots, k_1, k_0)_2$，$l$ 为 k 的比特长度，对于 X25519 算法 $l = 255$；P 为 Curve25519 曲线上一椭圆曲线点，其仿射坐标 x 值为 x_P；$a_{24} = (a+2)/4$，a 为式 2.6 中的常量

输出: 标量乘法结果 $Q = kP$ 的仿射坐标 x 值 x_Q

初始化：$X_{R_0} \leftarrow 1, Z_{R_0} \leftarrow 0, X_{R_1} \leftarrow x_P, Z_{R_1} \leftarrow 1, s \leftarrow 0$
for $i \leftarrow l-1$ *downto 0* **do**
  $s \leftarrow s \otimes k_i$
  $X_{R_0}, X_{R_1} \leftarrow cswap(s, X_{R_0}, X_{R_1})$
  $Z_{R_0}, Z_{R_1} \leftarrow cswap(s, Z_{R_0}, Z_{R_1})$
  $s \leftarrow k_i$
  $A \leftarrow X_{R_0} + Z_{R_0}; B \leftarrow X_{R_0} - Z_{R_0}$
  $C \leftarrow X_{R_1} + Z_{R_1}; D \leftarrow X_{R_1} - Z_{R_1}$
  $C \leftarrow C \times B; D \leftarrow D \times A$
  $X_{R_1} \leftarrow D + C; X_{R_1} \leftarrow X_{R_1}^2$
  $Z_{R_1} \leftarrow D - C; Z_{R_1} \leftarrow Z_{R_1}^2; Z_{R_1} \leftarrow x_P \times Z_{R_1}$
  $A \leftarrow A^2; B \leftarrow B^2$
  $X_{R_0} \leftarrow A \times B$
  $A \leftarrow A - B$
  $Z_{R_0} \leftarrow a_{24} \times A; Z_{R_0} \leftarrow Z_{R_0} + B; Z_{R_0} \leftarrow Z_{R_0} \times A$
end
$X_{R_0}, X_{R_1} \leftarrow cswap(s, X_{R_0}, X_{R_1})$
$Z_{R_0}, Z_{R_1} \leftarrow cswap(s, Z_{R_0}, Z_{R_1})$
$Z_{R_0} \leftarrow Z_{R_0}^{-1}$
$x_Q \leftarrow X_{R_0} \times Z_{R_0}$
return x_Q

述的 (X, Z) 坐标进行计算，其中计算点加和倍点采用式 2.12，在 (X, Z) 坐标下完成主要计算后再转换至仿射坐标从而得到最终的结果。算法 2.6 中所用到的 $cswap$ 操作用于条件交换，以 $cswap(s, X_{R_0}, X_{R_1})$ 为例，s 为 1 则输出为 X_{R_1}, X_{R_0}，否则输出 X_{R_0}, X_{R_1}，该操作的实现无需使用分支操作，如 C 语言中的 if-else 语句，从而确保执行时间是恒定的。

对于 Ed25519 数字签名算法中所涉及的标量乘法运算，其计算思路与本文 2.2.3 所述的算法 2.3 相似，此处不再赘述。

2.3.4 协议层运算

X25519 密钥协商算法由 Bernstein 于 2006 年提出[44]，IETF（Internet Engineering Task Force）于 2016 年发布的 RFC 7748 标准文档[19] 除了包含了 128 比特安全性的 X25519 密钥协商算法还

算法 2.7: X25519-KeyGen 密钥生成

输出: 公钥 pk 和私钥 sk

随机生成 32 字节的私钥 sk

$(x_A, y_A) \leftarrow sk \cdot P$; /* 固定点标量乘，P 为 X25519 基点 */

$pk \leftarrow x_A$

return pk

算法 2.8: X25519-Derive 密钥协商

输入: 本端私钥 sk 和对端公钥 pk_{peer}

输出: 共享密钥 key_{shared}

$key_{shared} \leftarrow \text{MontLadder}(sk, pk_{peer})$; /* 使用蒙哥马利阶梯算法即算法 2.6 计算非固定点标量乘 */

return key_{shared}

算法 2.9: Ed25519-Sign 签名生成

输入: 待签名消息 M；签名者的公私钥对 (pk, sk)

输出: 数字签名对 (r, s)

$(a_l, a_h) \leftarrow \text{SHA512}(sk)$; /* a_l 为 SHA512 结果的低 32 字节，a_h 为 SHA512 结果的高 32 字节 */

$k \leftarrow \text{SHA512}(a_h||M) \mod n_{25519}$; /* n_{25519} 为椭圆曲线加法群的阶 */

$R \leftarrow k \cdot P$; /* 固定点标量乘，P 为基点 */

$r \leftarrow \text{Encode}(R)$; /* 将椭圆曲线点 R 编码为字节序列 */

$h \leftarrow \text{SHA512}(r||pk||M) \mod n_{25519}$

$s \leftarrow k + h \cdot a_l \mod n_{25519}$

return (r, s)

包含了 224 比特安全性的 X448 密钥协商算法。Ed25519 数字签名算法由 Bernstein 等人于 2012 年提出[46]，IETF 于 2017 年发布的 RFC 8032 标准文档[53] 包含了 128 比特安全性的 Ed25519 和 224 比特安全性的 Ed448 数字签名算法。IETF 于 2018 年在 RFC 8446 标准文档[4] 中发布了 TLS（Transport Layer Security）1.3 标准，其中推荐 X25519 密钥协商算法与 Ed25519 数字签名算法为标准算法。

X25519 密钥协商算法主要包含两个步骤：密钥生成和密钥协商。X25519 密钥生成算法用于生成公私钥对，见算法 2.7，其中核心计算是固定点标量乘 $sk \cdot P$，sk 为 256 位整数，P 为 X25519 对应的椭圆曲线加法群的基点。X25519 密钥协商算法用于根据本端私钥和对端公钥计算双方的共享密钥，见算法 2.8，其中核心计算是非固定点标量乘，使用蒙哥马利阶梯算法即算法 2.6 进行计算，并且只需要得到标量乘结果点的仿射坐标 x 值。

Ed25519 数字签名算法主要包含两个步骤：签名生成和签名验证。Ed25519 签名生成算法用于根据签名者的公私钥对 (pk, sk) 对待签名消息 M 进行数字签名，见算法 2.9，其中核心计算是固定点标量乘 $k \cdot P$，k 为 256 位整数，P 为椭圆曲线加法群的基点。Ed25519 签名验证算法用于根据签名者的公钥 pk 去验证消息 M' 的数字签名是否是 (r', s')，见算法 2.10，其中核心的计算是双点标量乘 $s \cdot P + h \cdot Q$，s' 与 h 为 256 位整数，P 为基点，Q 为一椭圆曲线点。

算法 2.10: Ed25519-Verify 签名验证

输入: 待验证消息 M；待验证签名对 (r, s)；签名者的公钥 pk

输出: 签名验证结果：成功或失败

if $s \notin [1, n_{25519} - 1]$ **then**
  return 失败
end
$Q \leftarrow \text{Decode}(pk)$; /* 将公钥解码为椭圆曲线点 Q */
if Q 不是椭圆曲线上的点 **then**
  return 失败
end
$h \leftarrow \text{SHA512}(r||pk||M) \mod n_{25519}$
$Q \leftarrow -Q$; /* 椭圆曲线点取负 */
$R \leftarrow s \cdot P + h \cdot Q$; /* 双点标量乘，P 为基点 */
$r' \leftarrow \text{Encode}(R)$; /* 将椭圆曲线点 R 编码为字节序列 */
if $r' = r$ **then**
  return 成功
else
  return 失败
end

2.4 ML-KEM

NIST FIPS 203 文件[11] 正式制定了 ML-KEM 标准算法，其全称为 Module-Lattice-based Key-Encapsulation Mechanism，密钥封装机制（KEM）是用于为通讯双方确定一个共享密钥的一系列算法。ML-KEM 基于 CRYSTALS-Kyber 算法进行构造，安全性基于 Module Learning With Errors（MLWE）问题，提供了三个参数集，不同的参数在安全强度和算法性能上有着不同的表现。

ML-KEM 与 NIST 后量子标准化项目第三轮的 CRYSTALS-Kyber 算法存在一些区别，具体为：

1. Kyber 中共享密钥的长度不是固定的，而 ML-KEM 中则为固定的 256 比特，并且可直接用于对称加密或用于推导对称加密密钥。
2. ML-KEM 中的 Encaps 和 Decaps 算法所使用的 FO 变换（Fujisaki-Okamoto transform）与 Kyber 中略微不同，即 ML-KEM.Encaps 中推导共享密钥时不再包括密文的哈希。
3. Kyber 的 Encpas 中先对随机值 m 做了哈希运算，而 ML-KEM 则无需该步骤。
4. ML-KEM 新增了部分输入合法性检查，比如 ML-KEM.Encaps 需检查字节序列解码得到的整数是否在规定范围内。

2.4.1 NTT 多项式乘法

NTT 多项式乘法是 ML-KEM 中的核心运算之一，其时间复杂度为 $O(nlogn)$，用于加速多项式环 R_q 上的多项式乘法运算，其中 $R_q = \mathbb{Z}_q[X]/(X^n+1)$。下文的符号表示和描述方式与 FIPS 203[11] 保持一致。环 R_q 与 T_q 同构，T_q 是 128 个 $\mathbb{Z}_q$ 二次扩域的集合，NTT 算法可高效地计算这两个环的同构变换。对于 $f \in R_q$，对其执行 NTT 变换操作，即 $\hat{f} := \mathrm{NTT}(f)$，其中 $\hat{f} \in T_q$ 且称其为 f 的 NTT 域表示。上述两个环同构意味着 $f \times_{R_q} g = \mathrm{NTT}^{-1}(\hat{f} \times_{T_q} \hat{g})$，其中 $\times_{R_q}$ 和 $\times_{T_q}$ 分别表示 R_q 和 T_q 中的乘法。考虑到 T_q 是 128 个环的乘积，每个由阶为 1 的多项式组成，因此 $\times_{T_q}$ 操作比 $\times_{R_q}$ 操作更高效。

因为环 R_q 和 T_q 具有 $\mathbb{Z}_q$ 上的向量空间结构，因此常用 $\mathbb{Z}_q^n$ 这一抽象数据结构来表示环中的元素。因此用于表示 NTT 和 INTT 输入输出的数据结构是长度为 n 的模 q 整数数组，这一数组也用于表示环 R_q 和 T_q 中的元素。

对于 ML-KEM，q 为素数且值为 $3329 = 2^8 \cdot 13 + 1$，$n = 256$。$\mathbb{Z}_q$ 上存在 256 次本原根，不存在 512 次本原根，$\zeta = 17 \in \mathbb{Z}_q$ 为一个 256 次本原根，且 $\zeta^{128} \equiv -1$。多项式 $X^{256}+1$ 可分解为 128 个阶为 2 的多项式模 q，即

$$X^{256} + 1 = \sum_{k=0}^{127} (X^2 - \zeta^{2\mathrm{BitRev}_7(k)+1}),$$

其中 BitRev_7 表示无符号 7 位整数的比特逆。因此，$\mathbb{Z}_q$ 与 T_q 同构，且

$$T_q := \prod_{k=0}^{127} \mathbb{Z}_q[X]/(X^2 - \zeta^{2\mathrm{BitRev}_7(k)+1}).$$

因此 $f \in R_q$ 的 NTT 域表示 $\hat{f} \in T_q$ 可表示为向量的形式：

$$\hat{f} := (f \mod (X^2 - \zeta^{2\mathrm{BitRev}_7(0)+1}), \cdots, f \mod (X^2 - \zeta^{2\mathrm{BitRev}_7(127)+1})).$$

也可表示为 256 个整数模 q，即

$$f \mod (X^2 - \zeta^{2\mathrm{BitRev}_7(i)+1}) = \hat{f}[2i] + \hat{f}[2i+1]X,$$

其中 $i = 0, \cdots, 127$。

2.4.2 K-PKE 方案

K-PKE 方案是 ML-KEM 的基本组件，其包含三个算法，分别是密钥生成（K-PKE.KeyGen）、加密（K-PKE.Encrypt）、解密（K-PKE.Decrypt），分别见算法 2.11 ~ 算法 2.13，具体参数的取

表 2.1　ML-KEM 方案参数集；后三列表示分别表示封装密钥、解封装密钥和密文的长度，单位为字节；所有 ML-KEM 方案的共享密钥长度均为 32 字节

	n	q	k	η_1	η_2	d_u	d_v	随机数安全强度	封装密钥	解封装密钥	密文
ML-KEM-512	256	3329	2	3	2	10	4	128	800	1632	768
ML-KEM-768	256	3329	3	2	2	10	4	192	1184	2400	1088
ML-KEM-1024	256	3329	4	2	2	11	5	256	1568	3168	1568

值见表 2.1。

K-PKE.KeyGen 无需输入；计算过程需要随机数；输出为加密和解密密钥 $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}})$，其中加密密钥可以公开，解密密钥则不能公开。K-PKE.Encrypt 的输入包括加密密钥 $\mathsf{ek}_{\mathsf{PKE}}$、明文 m、随机数 r；输出为密文 c。K-PKE.Decrypt 的输入包括解密密钥 $\mathsf{dk}_{\mathsf{PKE}}$ 和密文 c；输出为明文 m。

2.4.3　基于 K-PKE 的 ML-KEM 方案

ML-KEM 方案基于 K-PKE 并使用 FO 变换[54,55] 构造，包括三个算法，分别是密钥生成（ML-KEM.KeyGen）、封装（ML-KEM.Encaps）和解封装（ML-KEM.Decaps），分别见算法 2.14 ~ 算法 2.15，具体参数的取值见表 2.1，三个不同的参数集分别命名为 ML-KEM-512、ML-KEM-768 和 ML-KEM-1024。

ML-KEM.KeyGen的核心子程序是K-PKE.KeyGen，ML-KEM 的封装密钥即为 K-PKE 的加密密钥，ML-KEM 的解封装密钥包括 K-PKE 的解密密钥、封装密钥、封装密钥的哈希值和 32 字节伪随机数。

ML-KEM.Encaps的输入为封装密钥；计算过程需要随机数；输出为密文和共享密钥。ML-KEM.Encaps 需先验证输入的 ek 是否是合法值，如果不是合法值算法终止运行。ML-KEM.Encaps 的核心子程序是K-PKE.Encrypt，其用于将随机值 m 加密成密文 c。

ML-KEM.Decaps的输入为解封装密钥和密文；输出为共享密钥。该算法也对输入进行合法性验证，如果不是合法值算法终止运行。ML-KEM.Decaps 算法先从 ML-KEM 的解封装密钥 dk 中解析出 K-PKE 的加密/解密密钥、哈希值 h、随机值 z。解密密钥用于对密文解密从而得到 m'，然后对 m' 重新加密从而得到 c'，再判断 c 与 c' 是否相同。

算法 2.11: K-PKE.KeyGen 密钥生成

输出: 加密/解密密钥 $(\mathsf{ek_{PKE}}, \mathsf{dk_{PKE}})$

$d \leftarrow \{0,\cdots,255\}^{32}$

$(\rho,\sigma) \leftarrow \mathrm{G}(d)$

$\hat{\mathbf{A}} \leftarrow \mathrm{GenMatrixA}(\rho)$

$\mathbf{s},\mathbf{e} \leftarrow \mathrm{SampleVec}_{\eta_1}(\sigma)$

$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}) + \mathrm{NTT}(\mathbf{e})$

$\mathsf{ek_{PKE}} \leftarrow \mathrm{ByteEncode}_{12}(\hat{\mathbf{t}}) \| \rho$

$\mathsf{dk_{PKE}} \leftarrow \mathrm{ByteEncode}_{12}(\hat{\mathbf{s}})$

return $(\mathsf{ek_{PKE}}, \mathsf{dk_{PKE}})$

算法 2.12: K-PKE.Decrypt 解密

输入: 解密密钥 $\mathsf{dk_{PKE}}$；密文 c

输出: 32 字节消息 m

$c_1, c_2 \leftarrow c[0 : 32d_u k : 32(d_u k + d_v)]$

$\mathbf{u} \leftarrow \mathrm{Decompress}_{\mathrm{d_u}}(\mathrm{ByteDecode}_{\mathrm{d_u}}(c_1))$

$v \leftarrow \mathrm{Decompress}_{\mathrm{d_v}}(\mathrm{ByteDecode}_{\mathrm{d_v}}(c_2))$

$\hat{\mathbf{s}} \leftarrow \mathrm{ByteDecode}_{12}(\mathsf{dk_{PKE}})$

$w \leftarrow v - \mathrm{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathrm{NTT}(\mathbf{u}))$

$m \leftarrow \mathrm{ByteEncode}_1(\mathrm{Compress}_1(w))$

return m

算法 2.13: K-PKE.Encrypt 加密

输入: 加密密钥 $\mathsf{ek_{PKE}}$；32 字节消息 m；32 字节随机值 r

输出: 密文 c

$\hat{\mathbf{t}} \leftarrow \mathrm{ByteDecode}_{12}(\mathsf{ek_{PKE}}[0 : 384k])$

$\rho \leftarrow \mathsf{ek_{PKE}}[384k : 384k + 32]$

$\hat{\mathbf{A}} \leftarrow \mathrm{GenMatrixA}(\rho)$

$\mathbf{r} \leftarrow \mathrm{SampleVec}_{\eta_1}(r, 0)$

$\mathbf{e_1} \leftarrow \mathrm{SampleVec}_{\eta_2}(r, N)$

$e_2 \leftarrow \mathrm{SamplePoly}_{\eta_2}(r, 2N)$

$\hat{\mathbf{r}} \leftarrow \mathrm{NTT}(\mathbf{r})$

$\mathbf{u} \leftarrow \mathrm{NTT}^{-1}\left(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}\right) + \mathbf{e_1}$

$\mu \leftarrow \mathrm{Decompress}_1(\mathrm{ByteDecode}_1(m))$

$v \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mu$

$c_1 \leftarrow \mathrm{ByteEncode}_{\mathrm{d_u}}(\mathrm{Compress}_{\mathrm{d_u}}(\mathbf{u}))$

$c_2 \leftarrow \mathrm{ByteEncode}_{\mathrm{d_v}}(\mathrm{Compress}_{\mathrm{d_v}}(v))$

return $c \leftarrow (c_1 \| c_2)$

2.5 ML-DSA

NIST FIPS 204 文件[12] 正式制定了 ML-DSA 标准算法，其全称为 Module Lattice Digital Signature Algorithm，其基于 CRYSTALS-Dilithium 算法进行构造。ML-DSA 提供了三种不同的参数集，分别为 ML-DSA-44、ML-DSA-65 和 ML-DSA-87，具体的参数选择见表 2.2。ML-DSA 与 NIST 后量子标准化项目第三轮的 CRYSTALS-Dilithium 算法（3.1 版本）存在一些区别，具体为：

1. 对于 ML-DSA-65 和 ML-DSA-87，$\tilde{c}$ 的长度分别被增加到了 384 和 512 比特，tr 的长度被增加到 512 比特。
2. 在 Dilithium 中，对于确定性版本的签名算法，ρ' 通过伪随机函数以签名者的私钥和消息为输入生成；对于随机版本的签名算法，ρ' 是一个 512 位的随机数。在 ML-DSA 中，对于确定性版本的签名算法，ρ' 通过伪随机函数以签名者的私钥、消息和 256 位随机数 rnd 为输入生成；对于随机版本的签名算法，ρ' 是一个 256 位的随机数。

算法 2.14: ML-KEM.KeyGen

输出: 封装密钥 ek；解封装密钥 dk

$z \leftarrow \{0,\cdots,255\}^{32}$

$(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}) \leftarrow$K-PKE.KeyGen()

$\mathsf{ek} \leftarrow \mathsf{ek}_{\mathsf{PKE}}$

$\mathsf{dk} \leftarrow (\mathsf{dk}_{\mathsf{PKE}} \| \mathsf{ek} \| H(\mathsf{ek}) \| z)$

return $(\mathsf{ek}, \mathsf{dk})$

算法 2.15: ML-KEM.Encaps

输入: 封装密钥 ek

输出: 共享密钥 K；密文 c

$m \leftarrow \{0,\cdots,255\}^{32}$

$(K, r) \leftarrow G(m \| H(\mathsf{ek}))$

$c \leftarrow$K-PKE.Encrypt(ek, m, r)

return (K, c)

算法 2.16: ML-KEM.Decaps

输入: 密文 c；解封装密钥 dk

输出: 共享密钥 K

$\mathsf{dk}_{\mathsf{PKE}}, \mathsf{ek}_{\mathsf{PKE}} \leftarrow \mathsf{dk}[0 : 384k : 768k + 32]$

$\mathsf{h} \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$

$\mathsf{z} \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$

$m' \leftarrow$K-PKE.Decrypt($\mathsf{dk}_{\mathsf{PKE}}, c$)

$(K', r') \leftarrow G(m' \| h)$

$\bar{K} \leftarrow J(z \| c, 32)$

$c' \leftarrow$K-PKE.Encrypt($\mathsf{ek}_{\mathsf{PKE}}, m', r'$)

if $c \neq c'$ **then**

  $K' \leftarrow \bar{K}$

end

return K'

2.5.1 NTT 多项式乘法

ML-DSA 与 ML-KEM 的 NTT 多项式乘法类似，区别在于模数 q 的选取不同。对于 ML-DSA，模数 $q = 2^{23} - 2^{13} + 1 = 8380417$，$n = 256$。$\mathbb{Z}_q$ 中存在 512 次本原根，$\zeta = 1753 \in \mathbb{Z}_q$ 为一个 512 次本原根。对于 $w \in R_q$，对 w 执行 NTT 变换可表示为：

$$\mathrm{NTT}(w) = (w(\zeta_0), w(-\zeta_0), \cdots, w(\zeta_{127}), w(-\zeta_{127})) \in T_q,$$

其中，$\zeta_i = \zeta^{\mathrm{brv}(128+i)}$，brv 表示无符号 8 比特整数的比特逆。

2.5.2 ML-DSA 方案

ML-DSA 数字签名方案包含三个算法，分别为 ML-DSA.KeyGen（算法 2.20）、ML-DSA.Sign（算法 2.22）和 ML-DSA.Verify（算法 2.21）。ML-DSA 的安全性基于两个困难问题，分别是 Module Learning With Errors（MLWE）和 SelfTargetMSIS 问题。

ML-DSA.KeyGen 无需输入；输出为公钥和私钥。该算法先使用随机数生成器生成一个 256 位的随机数 ζ，然后使用扩展输出函数生成其他的随机数，具体为：

1. ρ，用于生成多项式矩阵 $\mathbf{A} \in R_q^{k \times l}$。
2. ρ'，用于生成多项式向量 $\mathbf{s}_1 \in R_q^l$ 和 $\mathbf{s}_2 \in R_q^k$，其中多项式系数范围为 $[-\eta, \eta]$。
3. K，用于签名的计算。

ML-DSA.KeyGen 中的核心运算是 $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$。

表 2.2 ML-DSA 方案参数集；后三行分别表示私钥、公钥和签名的长度，单位为字节

参数	ML-DSA-44	ML-DSA-65	ML-DSA-87
q：模数	8380417	8380417	8380417
d：$\mathbf{t}$ 中丢掉的位数	13	13	13
τ：c 中 ±1 的数量	39	49	60
λ：$\tilde{c}$ 的碰撞强度	128	192	256
γ_1：$\mathbf{y}$ 的系数范围	2^{17}	2^{19}	2^{19}
γ_2：LowBits 输出范围	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k,ℓ)：$\mathbf{A}$ 的维度	$(4,4)$	$(6,5)$	$(8,7)$
η：私钥范围	2	4	2
$\beta=\tau\cdot\eta$：$c\mathbf{s_i}$ 的系数范围	78	196	120
ω：$\mathbf{h}$ 中 1 的数量最大值	80	55	75
挑战熵	192	225	257
签名过程中的平均重复次数	4.25	5.1	3.85
NIST 安全强度范围	2	3	5
私钥长度	2528	4000	4864
公钥长度	1312	1952	2592
签名长度	2420	3293	4595

ML-DSA.Sign 的输入为私钥 sk 和消息 M；输出为签名 σ。ML-DSA.Verify 的输入为公钥 pk、消息 M 和签名 σ；输出为布尔值（1 表示验证成功；0 表示验证失败）。

2.6 Saber

Saber 方案[56,57] 的安全性基于 MLWR（Module Learning With Rounding）问题[58]，Saber 方案曾入选 NIST 后量子密码标准化项目最终轮次，是四个 KEM 候选算法之一。IND-CCA 安全的 Saber.KEM 方案基于 IND-CPA 安全的 Saber.PKE 方案并使用 FO 变换以及一些对称密码学原语进行构造。Saber.PKE 公钥加密方案包含三个算法，分别是密钥生成 Saber.PKE.KeyGen（算法 2.17）、加密 Saber.PKE.Enc（算法 2.19）和解密 Saber.PKE.Dec（算法 2.18）。

Saber 方案中用到了两个模数，分别是 p 和 q，这两个模数的值分别为 2^{ϵ_p} 和 2^{ϵ_q}，因此 Saber 方案存在一些优势：（1）无需进行噪声采样和拒绝采样，而基于 MLWE 困难问题的 ML-KEM 则需要这两个步骤；（2）计算多项式乘法时无需计算复杂的模约减。

Saber 方案提供了三个不同的安全参数集，分别命名为 LightSaber、Saber 和 FireSaber，模格的维度 l 分别为 2、3 和 4。考虑到本文主要针对 Saber.PKE 方案中的核心运算进行研究，因

算法 2.17: Saber.PKE.KeyGen

输出: 公钥和私钥 (pk, sk)

$\rho, \sigma \leftarrow \{0, \cdots, 255\}^{32}$

$\mathbf{A} \leftarrow \text{GenMatrixA}(\rho)$

$\mathbf{s} \leftarrow \text{SampleVec}_{\mu}(\sigma)$

$\mathbf{b} \leftarrow (\mathbf{A}^T\mathbf{s} + \mathbf{h}) >> (\epsilon_q - \epsilon_p)$

return $(pk := (\mathbf{b}, \rho), sk := \mathbf{s})$

算法 2.18: Saber.PKE.Dec

输入: 私钥 $sk := \mathbf{s}$；密文 $c := (c_m \| \mathbf{b}')$

输出: 32 字节消息 m'

$v \leftarrow \mathbf{b}'^T(\mathbf{s} \mod p)$

$m' \leftarrow ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \mod p) >> (\epsilon_p - 1)$

return m'

算法 2.19: Saber.PKE.Enc

输入: 公钥 $pk := (\mathbf{b}, \rho)$；32 字节消息 m；32 字节随机值 r

输出: 密文 c

$\mathbf{A} \leftarrow \text{GenMatrixA}(\rho)$

$\mathbf{s}' \leftarrow \text{SampleVec}_{\mu}(r)$

$\mathbf{b}' \leftarrow (\mathbf{A}\mathbf{s}' + \mathbf{h}) >> (\epsilon_q - \epsilon_p)$

$v' \leftarrow \mathbf{b}^T(\mathbf{s}' \mod p)$

$c_m \leftarrow ((v' + h_1 - 2^{\epsilon_p - 1} m) \mod p) >> (\epsilon_p - \epsilon_T)$

return $c := (c_m \| \mathbf{b}')$

此没有给出 Saber.KEM 方案的算法描述，对于更多关于 Saber 方案的细节和参数选取，请参考文献[56,57]。

多项式乘法 虽然 Saber 方案的模数 $q = 2^{13}$ 在一定程度上简化了模约减的计算，但这一参数选取也导致 Saber 无法使用性能优异的 NTT 多项式乘法。因此，Saber 的参考实现中使用了 Toom-Cook 和 Karatsuba 结合的算法来计算多项式乘法。除 Saber 方案以外，NTRU 和 LAC 方案的多项式环也无法直接使用 NTT 多项式乘法，我们统称这类多项式环为 NTT 不友好的多项式环。2021 年，Chung 等人[37] 为 Saber、NTRU 和 LAC 方案适配了 NTT 多项式乘法，并且相比于其他实现方法具有明显的性能优势。

2.7 本章小结

本章介绍了本工作所需的符号定义、商密 SM2 数字签名算法、X25519 与 Ed25519 算法、ML-KEM 方案、ML-DSA 方案和 Saber 方案，作为下文具体优化实现的基础背景。

算法 2.20: ML-DSA.KeyGen 密钥生成

输出: 公钥和私钥 (pk, sk)

$\zeta \leftarrow \{0,1\}^{256}$

$(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} \leftarrow \mathrm{H}(\zeta)$

$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \mathrm{ExpandA}(\rho)$

$(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k \leftarrow \mathrm{ExpandS}(\rho')$

$\mathbf{t} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$

$(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \mathrm{Power2Round}(\mathbf{t}, d)$

$pk \leftarrow \mathrm{pkEncode}(\rho, \mathbf{t}_1)$

$tr \in \{0,1\}^{512} \leftarrow \mathrm{H}(pk)$

$sk \leftarrow \mathrm{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

return (pk, sk)

算法 2.21: ML-DSA.Verify 验证

输入: 公钥 pk；消息 M；签名 σ

输出: 成功或失败

$(\rho, \mathbf{t_1}) \leftarrow \mathrm{pkDecode}(pk)$

$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \mathrm{sigDecode}(\sigma)$

if $\mathbf{h} = \perp$ **then return** 失败 **end**

$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \mathrm{ExpandA}(\rho)$

$tr \in \{0,1\}^{512} \leftarrow H(pk)$

$\mu \in \{0,1\}^{512} \leftarrow \mathrm{H}(tr \| M)$

$(\tilde{c_1}, \tilde{c_2}) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256} \leftarrow \tilde{c}$

$c \leftarrow \mathrm{SamplelnBall}(\tilde{c})$

$\mathbf{w}_\mathbf{t}' \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{z}) - \mathrm{NTT}(c) \circ \mathrm{NTT}(\mathbf{t_1} 2^\mathbf{d}))$

$\mathbf{w}_1' \leftarrow \mathrm{UseHint}\left(\mathbf{h}, \mathbf{w}_\mathbf{t}'\right)$

return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **&&** $[\![\tilde{c} = \mathrm{H}\left(\mu \| \mathbf{w}_1'\right)]\!]$ **&&** $[\![\mathbf{h}$ 中 *1* 的数量 $\leq \omega]\!]$

算法 2.22: ML-DSA.Sign 签名生成

输入: 私钥 sk；消息 M

输出: 签名 σ

$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow \mathrm{skDecode}(sk)$

$\hat{\mathbf{s_1}}, \hat{\mathbf{s_2}}, \hat{\mathbf{t_0}} \leftarrow \mathrm{NTT}(\mathbf{s_1}), \mathrm{NTT}(\mathbf{s_2}), \mathrm{NTT}(\mathbf{t_0})$

$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \mathrm{ExpandA}(\rho)$

$\mu \in \{0,1\}^{512} \leftarrow \mathrm{H}(tr \| M)$

$rnd \leftarrow \{0,1\}^{256}, \kappa \leftarrow 0, (\mathbf{z}, \mathbf{h}) \leftarrow \perp$

$\rho' \in \{0,1\}^{512} \leftarrow \mathrm{H}(K \| rnd \| \mu)$

while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**

  $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell \leftarrow \mathrm{ExpandMask}(\rho', \kappa)$

  $\mathbf{w} \leftarrow \mathrm{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathrm{NTT}(\mathbf{y}))$

  $\mathbf{w}_1 \leftarrow \mathrm{HighBits}(\mathbf{w})$

  $\tilde{c} \in \{0,1\}^{2\lambda} \leftarrow \mathrm{H}(\mu \| \mathbf{w}_1)$

  $(\tilde{c_1}, \tilde{c_2}) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256} \leftarrow \tilde{c}$

  $c \leftarrow \mathrm{SamplelnBall}(\tilde{c_1}), \hat{c} \leftarrow \mathrm{NTT}(c)$

  $\mathbf{z} \leftarrow \mathbf{y} + \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s_1}})$

  $\mathbf{r}_0 \leftarrow \mathrm{LowBits}\left(\mathbf{w} - \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s_2}})\right)$

  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ 或 $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**

    $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$

  else

    $\mathbf{h} \leftarrow \mathrm{MakeHint}(-\mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}_0}), \mathbf{w} - \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s_2}}) + \mathrm{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}_0}))$

    if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ **或**$\mathbf{h}$ 中 *1* 的数量大于ω **then**

      $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$

    end

  end

  $\kappa \leftarrow \kappa + \ell$

end

return $\sigma \leftarrow \mathrm{sigEncode}(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h})$

第三章　商密 SM2 在国产华为鲲鹏处理器上的优化实现研究

椭圆曲线密码算法因其密钥长度短、计算效率高等优势，逐渐取代 RSA 成为各种网络协议中使用最广泛的公钥密码算法。商密标准 SM2 数字签名算法是我国自主制定的 ECC 算法标准，于 2010 年发布，旨在保障我国政府和企业的信息安全需求。近年来，我国加大了对国产处理器的研发与推广力度，基于 ARMv8-A 架构的华为鲲鹏 ARM 处理器便是其中的典型代表。尽管 SM2 的优化实现在 x86-64 指令集上已有充分研究，但针对国产华为鲲鹏 ARM 处理器的研究仍然相对不足。为解决 SM2 在国产 ARM 处理器上性能欠佳的问题，本章聚焦于我国商用密码标准 SM2 数字签名算法在华为鲲鹏 ARM 处理器上的性能优化，涵盖从底层有限域运算、标量乘法运算到协议层操作的全面优化。最终的性能测试表明，SM2 签名性能与验签性能分别是目前已知最快实现的 8.7 倍和 3.5 倍。这在一定程度上解决了 SM2 数字签名算法在国产 ARM 服务器上性能不佳的问题，并能有效推动 SM2 在国产 ARM 服务器上的应用和普及。

3.1　引言

公钥密码算法中的密钥协商协议和数字签名协议被广泛应用于互联网中安全信道的建立和身份认证[59]。1985 年，由 Koblitz 和 Miller 共同提出的椭圆曲线公钥密码算法（Elliptic Curve Cryptography, ECC）由于其密钥长度短、计算效率高等优势[42], 逐渐取代 RSA 算法成为 TLS[3,4]、SSH[5–8]、IPSec[9] 等网络协议中应用最为广泛的公钥密码算法。

商密标准 SM2 是我国自主制定的 ECC 算法标准，该标准于 2010 年发布[40]，旨在保障我国政府和企业的信息安全需求。SM2 采用了我国自主研发的椭圆曲线参数，具有自主知识产权，并提供了数字签名、密钥交换和公钥加密等功能，适用于各种信息安全领域，如电子认证、安全通信和电子支付等。在国际上，SM2 作为中国的国家密码算法标准，得到了国际社会的关注和认可，SM2 数字签名算法于 2018 年入选 ISO/IEC 标准[41]。越来越多的密码库开始支持 SM2 算法，例如 OpenSSL[60]、GmSSL[61] 和 Botan[62] 都已经支持 SM2 算法，并不断更新和优化相关功能，使得 SM2 在各种应用场景中得到更广泛的应用和推广。

SM2 与国际 ECC 标准 NIST P-256 算法[63] 具有相似的设计，它们的优化实现在 x86-64 架构上已经得到了充分的研究，主流的 OpenSSL 和 GmSSL 密码库均集成了它们的 x86-64 汇编优化实现[24]。近年来，ARM 架构服务器在我国的发展取得了显著进展，随着云计算、大数据、人工智能等领域的迅速发展，对于高性能、低功耗、灵活性强的服务器需求日益增加，ARM 架构服务器因其在能效比和成本效益上的优势逐渐受到重视。我国政府也积极推动 ARM 服务器的发展，提出了一系列政策措施支持本土 ARM 服务器产业的发展，鼓励企业进行技术研发和创新，

加大对 ARM 生态系统的支持力度。然而，SM2 在 ARM 架构服务器上的优化仍不充分，因此本章研究 SM2 数字签名算法在国产华为鲲鹏 ARM 处理器上的优化实现。鉴于华为鲲鹏处理器采用 ARMv8-A 架构，因此本章的研究也适用于诸如智能手机、平板电脑等场景中应用 ARMv8-A 架构的处理器。

本章贡献 为解决 SM2 在国产 ARM 处理器上性能欠佳的问题，本章针对 SM2 数字签名算法在华为鲲鹏 ARM 处理器上进行了深入的分析和优化，本章的优化从底向上覆盖了有限域运算、标量乘运算以及协议层运算，具体包括：

- 对于有限域运算，针对模 p_{sm2} 与模 n_{sm2} 乘法/平方运算，本章根据模数的数值特点提出了优化的蒙哥马利模乘 CIOS 方法，性能分别是之前实现的 1.3 倍和 1.2 倍。针对模 p_{sm2} 与模 n_{sm2} 求逆运算，本章推导并实现了更快的基于费马小定理的模逆算法，性能分别是之前实现的 2.1 倍和 1.5 倍。
- 对于固定点与非固定点标量乘运算，本章分别实现了宽度为 7 与 5 的窗口算法，性能分别是之前实现的 30.4 倍和 5.2 倍。对于固定点标量乘，用于标量乘的椭圆曲线点提前已知，因此预计算了 $37 \cdot 64 = 2368$ 个椭圆曲线点来减少固定点标量乘的计算开销。对于非固定点标量乘，预计算需要在运行时执行，因此只预计算 16 个椭圆曲线点。
- 对于签名生成过程中 s 的计算，本章提出了更快的计算方法，用一个模 n_{sm2} 加/减法来替换一个模 n_{sm2} 乘法从而加速签名生成的执行。
- 最终，本章将上述优化技术集成到了 OpenSSL 的性能测试框架中，并在华为云鲲鹏 920 处理器上进行了测试，测试表明 SM2 签名性能是之前实现的 8.7 倍，每秒可执行次数从 2413 次提升到了 21102 次；SM2 验签性能是之前实现的 3.5 倍，每秒可执行次数从 2526 次提升到了 8822 次。该工作可在一定程度上推进 SM2 在国产 ARM 服务器上的应用和普及。

3.2 优化方案的设计与实现

本节研究 SM2 数字签名算法在 ARMv8-A 架构上的优化实现。SM2 的优化实现在 Intel 处理器上已经得到了充分的研究，如主流的 OpenSSL 和 GmSSL 密码库都集成了 SM2 的 x86-64 汇编优化实现，但是 SM2 在 ARMv8-A 架构上的优化仍不充分。为解决 SM2 在 ARMv8-A 架构上性能欠佳的问题，本节提出了以下优化点：针对 SM2 的模 p_{sm2} 与模 n_{sm2} 乘法/平方运算，充分利用 p_{sm2} 与 n_{sm2} 的数值特点优化了蒙哥马利模乘；针对模 p_{sm2} 与模 n_{sm2} 求逆运算，推导并实现了更快的基于费马小定理的模逆算法；针对固定点与非固定点标量乘法，分别实现了宽度为 7 与 5 的窗口算法；针对签名生成过程中 s 的计算，用一个模 n_{sm2} 加/减法替换一个模 n_{sm2} 乘法。将上述优化技术集成到 OpenSSL（3.0.0-beta1）中后，在华为云鲲鹏 920 处理器上的测试

表明，SM2 签名性能提升 8.7 倍，每秒可执行次数从 2413 次提升到了 21102 次；SM2 验签性能提升 3.5 倍，每秒可执行次数从 2526 次提升到了 8822 次。在树莓派 4 上的测试表明，SM2 签名性能提升 9.7 倍，每秒可执行次数从 675 次提升到了 6534 次；SM2 验签性能提升 3.4 倍，每秒可执行次数从 777 次提升到了 2646 次。本节的实现无私钥相关的分支和内存访问，并且对预计算表的访问做了安全防护，可有效抵御简单能量攻击和缓存攻击。

3.2.1 ARMv8-A 架构简介

华为鲲鹏 920 处理器基于 ARMv8-A 架构，该架构是 ARM 公司面向高性能应用引入的 64 位架构，其同时支持 64 位和 32 位执行状态，即 AArch64 和 AArch32。在 AArch64 执行状态下，ARMv8-A 提供了一组 64 位指令集 A64，而 AArch32 对应的 32 位指令集则用于向前兼容 ARMv7-A 架构。

A64 指令集支持一系列 64 位的算术和逻辑运算，并支持较为丰富的乘法运算，比如乘法 MUL、乘法取负 MNEG、乘加 MADD 和乘减 MSUB 等指令。两个 64 位数相乘需要使用两条指令 {S,U}MULH 与 {S,U}MULL 才能得到完整的 128 位结果。ARMv8-A 架构提供了 31 个通用寄存器，其中 $w_0 \sim w_{30}$ 用于 32 位指令集，$x_0 \sim x_{30}$ 用于 64 位指令集。ARMv8-A 还支持 128 位单指令多数据（SIMD）指令集 NEON。每条 NEON 指令可同时处理 128 位数据，但是其只支持 32 位乘法运算，不支持 64 位乘法运算，并且不支持进位/借位处理。我们在实现 SM2 的有限域运算时采用 64 位的蒙哥马利 CIOS 方法，即算法 2.1。因为 NEON 最大只支持 32 位乘法运算，因此我们在实现 SM2 的有限域运算时不使用 NEON 指令集，仅使用 A64 指令集。

3.2.2 有限域运算优化：模乘/平方与模逆

本节考虑两种不同的模乘方法：第一种方法使用根据模数数值特征推导出的快速约减算法；第二种方法使用蒙哥马利模乘 CIOS 方法，并根据模数数值特征对其进行了优化。

模 p_{sm2} 时的快速约减算法 兰修文[64] 和何德彪教授等人[65] 分别推导了针对 SM2 模数 p_{sm2} 在 32 比特下的快速约减算法，本节所针对的 ARMv8-A 架构是 64 比特架构，因此本节针对 64 比特的快速约减算法进行推导。

针对两个大整数 $a, b \in F_{p_{sm2}}$，其多精度表示为：

$$a = \sum_{i=0}^{3} a_i \cdot 2^{64 \cdot i}, b = \sum_{i=0}^{3} b_i \cdot 2^{64 \cdot i} \tag{3.1}$$

这两个大整数的乘积表示为 $f = ab = \sum_{i=0}^{7} f_i \cdot 2^{64 \cdot i}$。利用快速约减算法对 f 进行约减的过程主

要利用以下模 p_{sm2} 同余式：

$$\begin{aligned}
2^{256} &= 2^{224} + 2^{96} - 2^{64} + 2^{0} \mod p_{sm2} \\
2^{320} &= 2^{224} + 2^{160} + 2^{32} + 2^{0} \mod p_{sm2} \\
2^{384} &= 2 \cdot 2^{224} + 2^{128} + 2^{96} + 2^{32} + 2^{0} \mod p_{sm2} \\
2^{448} &= 2 \cdot 2^{224} + 2^{192} + 2^{160} + 2 \cdot 2^{128} + 2^{96} - 2^{64} + 2 \cdot 2^{32} + 2 \cdot 2^{0} \mod p_{sm2},
\end{aligned} \tag{3.2}$$

将上述同余式代入乘积 f 的高半部分 $f_4 \sim f_7$ 中，然后合并同次数项即可得到快速约减算法。令 t_i, $i \in [0,3]$ 表示权重为 2^{64i} 的 128 比特系数。为方便表述，下文用 f_{il} 表示 f_i 左移 32 位，f_{ir} 表示 f_i 右移 32 位，则乘积 f 高半部分的约减过程为：

$$\begin{aligned}
t_3 &= f_3 + f_{4l} + f_{5l} + 2f_{6l} + f_7 + 2f_{7l} \\
t_2 &= f_2 + f_{5l} + f_6 + 2f_7 + f_{7l} \\
t_1 &= f_1 - f_4 + f_{4l} + f_{6l} + f_{7l} - f_7 \\
t_0 &= f_0 + f_4 + f_5 + f_{5l} + f_6 + f_{6l} + 2(f_7 + f_{7l})
\end{aligned} \tag{3.3}$$

通过进位传递将 $t_0 \sim t_2$ 中高于 64 位的部分累加到 t_3 之后，由于 t_3 的高 64 位 t_{3h} 的权重为 2^{256}，为得到有限域 $F_{p_{sm2}}$ 中唯一的结果，还需要再对 $t_{3h} \cdot 2^{256}$ 按照上述同余式进行多轮约减，直到 t_3 的比特长度不超过 64。

上述 64 比特的快速约减算法由于存在减法运算，故需对借位链进行处理，这使得实现颇为复杂。除此以外，该约减算法虽然没有使用乘法指令，但是加/减法次数过多，因此本节的实现并未采用该方法，下文提到的蒙哥马利模乘方法更为高效。

模 p_{sm2} 时的蒙哥马利模乘优化 与文献[24]中优化 NIST P-256 曲线所采用的思路类似，本节根据模数的数值特征，通过变换模数的表示将约减部分的乘法运算等价变换为加/减法，从而减少计算开销。SM2 模数 $p_{sm2} = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$ 的十六进制形式为：

$$\begin{aligned}
p_0 &= FFFFFFFFFFFFFFFF \\
p_1 &= FFFFFFFF\,0\,0\,0\,0\,0\,0\,0\,0 \\
p_2 &= FFFFFFFFFFFFFFFF \\
p_3 &= FFFFFFFEFFFFFFFF \\
p_{sm2} &= p_0 + p_1 2^{64\cdot 1} + p_2 2^{64\cdot 2} + p_3 2^{64\cdot 3}
\end{aligned} \tag{3.4}$$

考虑算法 2.1 第 10 行中的 mp_0，将 p_0 写为 $2^{64}-1$ 后发现：

$$mp_0 = m(2^{64}-1) = m2^{64} - m, \tag{3.5}$$

其中 $m2^{64}$ 等价于将 m 累加至权重为 2^{64} 的项中，$-m$ 等价于权重为 2^0 的项减 m，这样可将乘法运算转换为运行效率更高的加/减法运算。

基于上述观察，本节对 SM2 模数进行变换：

$$\begin{aligned} p_{sm2} &= (2^{256} + 2^{64} + 0) - (2^{224} + 2^{96} + 1) \\ &= (2^{256} - 2^{224}) + (2^{64} - 2^{96}) + (0 - 1) \\ &= (2^{64} - 2^{32})2^{192} + (1 - 2^{32})2^{64} + (-1), \end{aligned} \tag{3.6}$$

因此得到模数 p_{sm2} 的另一种表示方法：

$$\begin{aligned} &p_0 = -1, p_1 = 1 - 2^{32}, p_2 = 0, p_3 = 2^{64} - 2^{32} \\ &p_{sm2} = p_0 + p_1 2^{64\cdot 1} + p_2 2^{64\cdot 2} + p_3 2^{64\cdot 3}, \end{aligned} \tag{3.7}$$

从而可以将算法 2.1 第 10 和第 12 行的 mp_j（$j = 0, 1, 2, 3$）转换为：

$$\begin{aligned} mp_0 &= -m \\ mp_1 &= m(1 - 2^{32}) \\ &= -(m >> 32)2^{64} + m - (m << 32) \\ mp_2 &= 0 \\ mp_3 &= m(2^{64} - 2^{32}) \\ &= (m - (m >> 32))2^{64} - (m << 32) \end{aligned} \tag{3.8}$$

因此算法 2.1 第 10 和第 12 行的乘法运算 mp_j 全部被转换为了加/减法运算，其中算法 2.1 第 10-16 行的运算变换为：

$$\begin{aligned} &(C, S) = t_0 - m = t_0 - t_0 = 0 \\ &(C, S) = t_1 + m - (m << 32), \qquad t_0 = S \\ &(C, S) = t_2 - (m >> 32) + C, \qquad t_1 = S \\ &(C, S) = t_3 - (m << 32) + C, \qquad t_2 = S \\ &(C, S) = t_4 + m - (m >> 32) + C, \ t_3 = S. \end{aligned} \tag{3.9}$$

在 SM2 中算法 2.1 第 9 行用到的 $p'_0 = 1$，可节省一个乘法运算[64]，同时 $t_0 - m = t_0 - t_0 = 0$

算法 3.1: 蒙哥马利模乘 CIOS 方法中约减运算的 ARMv8-A 汇编实现，对应算法 2.1 第 9-17 行，模数为 p_{sm2}

输入: 算法 2.1 中 2-8 行计算得到的乘积：$t = \sum_{i=0}^{5} t_i 2^{64i}$

输出: 约减后的结果：$t = \sum_{i=0}^{4} t_i 2^{64i}$

```
1  MOV   m, t0;          /* m = t0*p'0 mod 2^64 (p'0 = 1) */
2  LSL   ml, m, #32;     /* m << 32 */
3  LSR   mr, m, #32;     /* m >> 32 */
4  ADDS  t0, t1, m;      /* t1 + m */
5  ADCS  t1, t2, 0;      /* t2 + C */
6  ADCS  t2, t3, 0;      /* t3 + C */
7  ADCS  t3, t4, m;      /* t4 + m + C */
8  ADCS  t4, t5, 0;      /* t5 + C */
9  SUBS  t0, t0, ml;     /* t0 - ml */
10 SBCS  t1, t1, mr;     /* t1 - mr - B (B 表示借位) */
11 SBCS  t2, t2, ml;     /* t2 - ml - B */
12 SBCS  t3, t3, mr;     /* t3 - mr - B */
13 SBC   t4, t4, 0;      /* t4 - B */
14 return t = sum_{i=0}^{4} t_i 2^{64i}
```

运算也可忽略。模平方的约减部分采用相同的优化思路。

算法 3.1 给出了约减部分 ARMv8-A 汇编实现伪代码，其中主要的实现技巧是先计算加法运算并处理进位链，然后再计算减法运算并处理借位链。

模 n_{sm2} 时的蒙哥马利模乘优化 SM2 椭圆曲线加法群的阶 n_{sm2} 不具备类似于模数 p_{sm2} 的数值特征，不便于设计快速约减算法，因此只考虑蒙哥马利模乘方法。这里仍基于算法 2.1 进行描述，但需将其中的模数 p_{sm2} 替换为阶 n_{sm2}，将 $p'_{sm2} = -p_{sm2}^{-1} \mod 2^{256}$ 替换为 $n_{sm2}{}' = -{n_{sm2}}^{-1} \mod 2^{256}$。

n_{sm2} 的十六进制形式为：

$$\begin{aligned} n_0 &= 53BBF40939D54123 \\ n_1 &= 7203DF6B21C6052B \\ n_2 &= FFFFFFFFFFFFFFFF \\ n_3 &= FFFFFFFEFFFFFFFF \\ n_{sm2} &= n_0 + n_1 2^{64\cdot 1} + n_2 2^{64\cdot 2} + n_3 2^{64\cdot 3}, \end{aligned} \tag{3.10}$$

其中 mn_2 与 mn_3 可使用上述相同的思路进行优化，即算法 2.1 的第 12 行（$j = 2, 3$）可变换为：

$m \cdot n_2 = m(2^{64}-1) = m2^{64} - m$ 和 $m \cdot n_3 = m(2^{64} - 2^{32}) = (m - (m >> 32))2^{64} - (m << 32)$。

算法 2.1 第 10 行的 mn_0 我们只需要计算该乘法的高 64 比特结果，原因是其满足条件：

$$\begin{aligned} &(t_0 + m \cdot n_0) \bmod 2^{64} \\ &= (t_0 + (t_0 \cdot (-n_0^{-1} \bmod 2^{64})) \cdot n_0) \bmod 2^{64} \\ &= (t_0 - t_0) \bmod 2^{64} \\ &= 0 \bmod 2^{64} \end{aligned} \tag{3.11}$$

即乘积中的低 64 比特恒为 0。但需要注意的是，此处需要分析两种情况：（1）$t_0 = 0$ 时，$m = 0$，此时 $t_0 + m \cdot n_0 = 0$；（2）$t_0 \neq 0$ 时，$t_0 + m \cdot n_0$ 的低 64 比特为 0，同时会向高 64 比特传递一个进位，形如 $(2^{64}-1)+1$ 的低 64 比特为 0，并向高 64 比特进位，该进位值记为 carry。

算法 2.1 第 12 行计算 mn_1 时，因为 n_1 没有形如 n_2 与 n_3 的数值特征，故需要计算完整的 128 比特乘法结果。综上，算法 2.1 第 10-16 行的运算变换为：

$$\begin{aligned} &C = \text{high}(m \cdot n_0) + \text{carry} \\ &(C, S) = t_1 + m \cdot n_1 + C, && t_0 = S \\ &(C, S) = t_2 - m + C, && t_1 = S \\ &(C, S) = t_3 + m - (m << 32) + C, && t_2 = S \\ &(C, S) = t_4 + m - (m >> 32) + C, && t_3 = S, \end{aligned} \tag{3.12}$$

其中 $\text{high}(m \cdot n_0)$ 表示乘积 $m \cdot n_0$ 的高 64 比特结果。上述公式的实现思路与算法 3.1 基本相同，因此不再给出汇编实现代码。

模逆优化 给定 $0 < a < p_{sm2}$，其中 p_{sm2} 为模数，为得到恒定时间的模逆实现，本节利用费马小定理计算 a 的乘法逆元 a^{-1}，即 $a^{-1} = a^{p_{sm2}-2} \mod p_{sm2}$。正如 Knuth Donald[66] 在其书籍第 4.6.3 章所述，给定一个指数 e 的加法链，计算 $x^e \mod p_{sm2}$ 时：平方运算相当于加法链中的乘 2 运算，乘法运算相当于加法链中的加法运算。比如指数 5 的加法链为 $1 \times 2 \times 2 + 1$，计算 x^5 的过程为：$a = x^2, a = a^2, a = a \times x$。

因此，模逆运算的效率取决于指数加法链的效率[67]，应尽可能减少计算过程中的模乘/平方运算，或者以模平方运算替换模乘运算，因为模平方的计算效率高于模乘。下文首先对现有实现方案进行分析，然后引出本节的优化方法。

对于模 p_{sm2} 与模 n_{sm2} 求逆，OpenSSL 中的 SM2 实现采用通用的模逆算法，使用了窗口宽度为 4 的窗口算法[68] 扫描指数 $p_{sm2} - 2$ 或 $n_{sm2} - 2$。在不考虑预计算开销的情况下，该方案消耗 $(256/4 - 1) \times 4 = 252$ 个模平方与 $256/4 - 1 = 63$ 个模乘运算。GmSSL 虽然不支持 ARMv8-A

架构，但是其提供了针对 SM2 模数的模逆优化实现[69]，该方案消耗 256 个模平方和 27 个模乘运算。Zhou Lu 等人[70] 针对 SM2 的模数 p_{sm2} 提出了更快的模逆算法，其消耗 255 个模平方和 15 个模乘运算。

而对于模 n_{sm2} 求逆，GmSSL 没有提供专门的优化方法; OpenSSL 对 NIST P-256 曲线的模阶求逆提供了一个优化方法[68]。NIST P-256 曲线的阶与 SM2 的类似，在计算 $Z^{-1} = Z^{n-2} \bmod n_{sm2}$ 时，$n_{sm2} - 2$ 的高 128 比特大部分为比特 1，具有较好的数值规律便于构造高效的加法链; 而低 128 比特则没有明显的数值规律，不方便构造加法链。因此，OpenSSL 只为高 128 比特构造了加法链，而低 128 比特使用宽度为 4 的窗口算法进行扫描。

我们观察到，模 p_{sm2} 求逆运算只需要用在雅可比投影坐标转仿射坐标的过程中用到，即: $(x, y) = (XZ^{-2}, YZ^{-3})$。

从上式可知，模 p_{sm2} 求逆运算主要用于计算 Z^{-2}与Z^{-3}。SM2 签名过程中 Z_A 的生成需要使用基点 G 和点 d_AG 的 x 和 y 值，因此需计算对应的 Z^{-2} 和 Z^{-3}。签名生成算法即算法 2.4 第 7 行与签名验证算法即算法 2.5 第 10 行中的计算仅涉及 x 值，因此只需要计算对应的 Z^{-2} 而不需要计算 Z^{-3}。

基于上述观察可知，模 p_{sm2} 求逆算法有两个使用场景：场景一为根据雅可比投影坐标求仿射坐标的 x 和 y 值，此时需要计算对应的 Z^{-2} 和 Z^{-3}；场景二为根据雅可比坐标求仿射坐标的 x 值，此时只需要计算对应的 Z^{-2}。本节的优化实现决定直接计算 $Z^{-2} = Z^{p-3} \mod p_{sm2}$。对于场景一，通过 $(Z^{-2})^2 \cdot Z$ 来构造 Z^{-3}，即通过一次求 Z^{-2} 运算、一次模平方和一次模乘来获得 Z^{-2} 和 Z^{-3}。原方法中使用 $(Z^{-1})^2$ 来构造 Z^{-2}，使用 $Z^{-2} \cdot Z^{-1}$ 来构造 Z^{-3}，共需计算一次求逆 (Z^{-1})、一次模平方和一次模乘运算。对于场景二，可以直接得到所需的 Z^{-2}，而原方法还需额外执行一次模平方运算。

本节 Z^{-2} 的计算方法见算法 3.2，算法注释中 S 表示模平方，M 表示模乘，该算法计算开销为 256 个模平方和 14 个模乘。与 Zhou Lu 等人[70]Z^{-1} 的计算开销相比，我们增加了一个模平方运算，减少了一个模乘运算，由于模平方运算快于模乘运算，并且在上述场景二中我们可以节省一次模平方运算，因此本节的优化方案具有更好的计算效率。

模 n_{sm2} 求逆用在签名生成算法即算法 2.4 的第 9 行，我们首先按照 OpenSSL 中 NIST P-256 曲线模阶求逆方法的思路，为 SM2 构造了求逆算法，见算法 3.3。然后提出了我们的优化方法，见算法 3.4，并对两者的计算开销进行了对比。

算法 3.3 先将 Z^1 至 Z^{15} 预计算至表 table 中，第 2 至 11 行用于构造 $n_{sm2} - 2$ 的高 128 比特，第 13 至 19 行使用宽度为 4 的窗口算法构建低 128 比特。其中表 exp[32] 以 4 比特为单位存储了 $n_{sm2} - 2$ 的低 128 比特。预计算阶段消耗 $8M + 8S$，构建高 128 比特消耗 $10M + 124S$，构建低 128 比特消耗 $29M + 128S$，其中 exp 有 3 个表项为 0，蒙哥马利域的转入和转出消耗 $2M$，总开销为 $49M + 260S$。

算法 3.2: 基于费马小定理计算 $Z^{-2} \mod p_{sm2}$

输入: Z 满足 $0 < Z < p_{sm2}$

输出: $Z^{-2} = Z^{p_{sm2}-3} \bmod p_{sm2}$

```
1  Z_2 ← Z^2 · Z;                                  /* 1S + 1M */
2  Z_4 ← Z_2^{2^2} · Z_2;                          /* 2S + 1M */
3  Z_6 ← Z_4^{2^2} · Z_2;                          /* 2S + 1M */
4  Z_12 ← Z_6^{2^6} · Z_6;                         /* 6S + 1M */
5  Z_24 ← Z_12^{2^12} · Z_12;                      /* 12S + 1M */
6  Z_30 ← Z_24^{2^6} · Z_6;                        /* 6S + 1M */
7  Z_31 ← Z_30^2 · Z;                              /* 1S + 1M */
8  Z_32 ← Z_31^2 · Z;                              /* 1S + 1M */
9  t ← (Z_31^{2^33} · Z_32)^{2^32} · Z_32;         /* 65S + 2M */
10 t ← (t^{2^32} · Z_32)^{2^32} · Z_32;            /* 64S + 2M */
11 t ← (t^{2^64} · Z_32)^{2^30} · Z_30;            /* 94S + 2M */
12 t ← t^{2^2};                                    /* 2S */
13 return t;                                       /* 总开销为: 256S + 14M */
```

算法 3.4 先预计算了 11 个常用值，包括 Z^{0b11}、Z^{0b101}、Z^{0b111} 等，其中 $0b$ 为二进制表示。然后使用这些预计算值来构建 $Z^{n_{sm2}-2}$。该算法的基本思想是：从 $n_{sm2}-2$ 的二进制表示中提炼出频繁出现的片段，如我们预计算值幂中的 $0b11$、$0b101$ 和 $0b111$，然后使用这些片段来构造 $n_{sm2}-2$。该算法的预计算部分开销为 $15M+65S$，计算部分开销为 $25M+192S$，蒙哥马利域的转入和转出消耗 $2M$，总开销为 $42M+257S$。相比于算法 3.3，本节提出的算法 3.4 节省了 7 次模乘与 3 次模平方运算。

3.2.3 标量乘法优化

在 OpenSSL 的 SM2 实现中，固定点与非固定点标量乘均使用蒙哥马利阶梯算法进行计算。Gueron Shay 等人将他们对 NIST P-256 曲线的优化实现[24] 集成到了 OpenSSL 中[71]。本节针对 SM2 中的标量乘运算进行优化，固定点标量乘法采用宽度为 7 的窗口算法，非固定点标量乘法采用宽度为 5 的窗口算法，标量使用 Booth 编码[72] 的变体。

Booth 编码变体 Booth 于 1951 年提出了 Booth 编码[72]，用于改进有符号数乘法。原始的 Booth 编码方案没有减少二进制表示中的非零数字的密度，但却是 NAF（Non-Adjacent Form）方法[73] 的基础。

以固定点标量乘法所用的宽度 $w=7$ 的窗口算法为例，我们采用 Booth 编码对标量 k 进行编码，在不改变 k 值的情况下，将 k 的每 7 个比特通过 2 的补码形式等价转换成有符号数表示，

算法 3.3: 模 n_{sm2} 求逆算法

输入: Z 满足 $0 < Z < n_{sm2}$

输出: $Z^{-1} = Z^{n_{sm2}-2} \bmod n_{sm2}$

预计算：$\mathsf{table}[i] \leftarrow Z^i, i = 1, \cdots, 15$
$t \leftarrow \mathsf{table}[15]^{2^4}$; /* $Z^{0\times F0}$ */
$t_2 \leftarrow t \cdot \mathsf{table}[14]$; /* $Z^{0\times FE}$ */
$t \leftarrow t \cdot \mathsf{table}[15]$; /* $Z^{0\times FF}$ */
$r \leftarrow t^{2^8}$, $t_2 \leftarrow r \cdot t_2$; /* $Z^{0\times FFFF}$ */
$r \leftarrow r \cdot t$; /* $Z^{0\times FFFE}$ */
$t \leftarrow r^{2^{16}}$, $t \leftarrow t \cdot t_2$; /* $Z^{0\times FFFFFFFE}$ */
$t_2 \leftarrow t \cdot \mathsf{table}[1]$; /* $Z^{0\times FFFFFFFF}$ */
$r \leftarrow t^{2^{32}}$, $r \leftarrow r \cdot t_2$; /* $Z^{0\times FFFFFFFEFFFFFFFF}$ */
$t \leftarrow t_2 \cdot \mathsf{table}[1]$; /* $Z^{0\times 100000000}$ */
$t \leftarrow t \cdot r$; /* $Z^{0\times FFFFFFFFFFFFFFFF}$ */
$r \leftarrow r^{2^{64}}$, $r \leftarrow r \cdot t$; /* 已构建好高 128 比特 */
将 $n_{sm2} - 2$ 低 128 位拆为 32 个 4 比特存至 $\mathsf{exp}[32]$ 中
// 构建低 128 比特
for $i \leftarrow 0$ *to* 31 **do**
    $r \leftarrow r^{2^4}$
    if $\mathsf{exp}[i] \neq 0$ **then**
        $r \leftarrow r \cdot \mathsf{table}[\mathsf{exp}[i]]$
    end
end
return $r = Z^{-1} \bmod n_{sm2}$

如果该 7 比特为负数，需向高字进位。例如，考虑对一个 14 比特的数字 s 应用 Booth 编码，s 的二进制表示为：$s = (s_{13}s_{12}\ldots s_1s_0)_2 = (0000111\ 1111000)_2$。因为窗口宽度为 7，故以 7 比特为单位对上述数字进行处理。以无符号数来看，低 7 比特的值为 120，高 7 比特的值为 7，该 14 比特的值为 $7 \cdot 2^7 + 120 = 1016$。将该数字进行 Booth 编码，低 7 比特视为有符号数补码，其值为 −8 需向高 7 比特进一位，高 7 比特的值变为 7 + 1 = 8，该 14 比特的值保持不变，仍然为 $8 \cdot 2^7 - 8 = 1016$。在具体实现时，我们将 −8 编码至一个 8 比特数中，其中最低位记录其符号，值为 1 表示负数、值为 0 表示正数，其余 7 比特记录绝对值 8，因此 −8 被编码为 $(0001\ 0001)_2$，+8 被编码为 $(0001\ 0000)_2$。

预计算技术及其存储方法 对于固定点标量乘法，因为宽度 $w = 7$，故共有 $\lceil 256/7 \rceil = 37$ 个窗口。又因为使用有符号数表示，故每个窗口包含 $2^6 = 64$ 个点。预计算表中的椭圆曲线点采用仿

算法 3.4: 优化版模 n_{sm2} 求逆算法

输入: Z 满足 $0 < Z < n_{sm2}$

输出: $Z^{-1} = Z^{n_{sm2}-2} \bmod n_{sm2}$

预计算: $Z^{0b1}, Z^{0b11}, Z^{0b101}, Z^{0b111}, Z^{0b1001}, Z^{0b1011}, Z^{0b1111}, Z^{10101}, Z^{0b11111}, Z_{x31} = Z^{0\times 7FFFFFFF}, Z_{x32} = Z^{0\times FFFFFFFF}$

$r \leftarrow Z_{x31}^{2^{33}} \cdot Z_{x32}$, $\quad r \leftarrow r^{2^{32}} \cdot Z_{x32}$

$r \leftarrow r^{2^{32}} \cdot Z_{x32}$, $\quad r \leftarrow r^{2^4} \cdot Z^{0b111}$

$r \leftarrow r^{2^3} \cdot Z^{0b1}$, $\quad r \leftarrow r^{2^{11}} \cdot Z^{0b1111}$

$r \leftarrow r^{2^5} \cdot Z^{0b1111}$, $\quad r \leftarrow r^{2^4} \cdot Z^{0b1011}$

$r \leftarrow r^{2^5} \cdot Z^{0b1011}$, $\quad r \leftarrow r^{2^3} \cdot Z^{0b1}$

$r \leftarrow r^{2^7} \cdot Z^{0b111}$, $\quad r \leftarrow r^{2^5} \cdot Z^{0b11}$

$r \leftarrow r^{2^9} \cdot Z^{0b101}$, $\quad r \leftarrow r^{2^7} \cdot Z^{0b10101}$

$r \leftarrow r^{2^5} \cdot Z^{0b10101}$, $\quad r \leftarrow r^{2^5} \cdot Z^{0b111}$

$r \leftarrow r^{2^4} \cdot Z^{0b111}$, $\quad r \leftarrow r^{2^6} \cdot Z^{0b11111}$

$r \leftarrow r^{2^3} \cdot Z^{0b101}$, $\quad r \leftarrow r^{2^{10}} \cdot Z^{0b1001}$

$r \leftarrow r^{2^5} \cdot Z^{0b111}$, $\quad r \leftarrow r^{2^5} \cdot Z^{0b111}$

$r \leftarrow r^{2^6} \cdot Z^{0b10101}$, $\quad r \leftarrow r^{2^2} \cdot Z^{0b1}$

$r \leftarrow r^{2^9} \cdot Z^{0b1001}$, $\quad r \leftarrow r^{2^5} \cdot Z^{0b1}$

return $r = Z^{-1} \bmod n_{sm2}$

射坐标 (x, y) 形式进行存储，因为仿射坐标-雅可比投影坐标混合加法运算的运行效率最快，具体计算方法参考文献[42] 中的算法 3.22。

其次，由于固定点运行前已知，而且 ARMv8-A 类型的设备存储资源较为充足，因此还可以通过预计算每个预计算点 $j \times G$ 的 2^{7i} 倍来减少标量乘法中的倍点运算，预计算表可表示为:

$$\mathsf{table}[i][j] = 2^{7i} \times (j \times G), \tag{3.13}$$

其中 $i = 0, 1, \cdots, 36$，$j = 0, 1, \cdots, 63$。每个点占用 64 字节的空间，一个子表包含 64 个点，共有 37 个子表，因此预计算表总大小为 $37 \times 64 \times 64/1024 = 148$KB。通过上述预计算技术，固定点标量乘法运算只需要 37 次点加运算，无需倍点运算。

对于非固定点标量乘法，因为预计算需在算法运行时执行，所以需要减少预计算量。本节采用窗口宽度为 $w = 5$ 的窗口算法，只需预计算 $2^4 = 16$ 个点，即 $i \times G$，$i = 0, 1, \cdots, 15$。通过上述预计算技术，在计算非固定点标量乘法时，预计算开销为 7 个点加和 8 个倍点运算，扫描窗口时需要 52 个点加和 255 次倍点运算，总开销为 59 个点加和 263 个倍点运算。

预计算表的访问顺序取决于标量 k，而缓存攻击可以通过探测缓存的访问模式来恢复出标量[74,75]。缓存的基本单位是缓存行（cache line），当代 CPU 的缓存行大小一般为 64 字节，而

每个预计算点的大小也为 64 字节。如果采用顺序存储的方法，缓存行的访问模式会泄漏标量信息。

因此在设计预计算表存储方式时需要使得访问不同预计算点时均访问相同的缓存行。本节采用与 OpenSSL 中 NIST P-256 曲线实现相同的方法：考虑 $w = 7$ 时每个子表的存储，每个子表包含 64 个点，将每个点的 64 字节拆成以字节为单位的 64 份，内存中的第 i 个 64 字节由 64 个点的第 i 个字节构成。以这样的方式，访问每个预计算点都需要访问相同的 64 个缓存行，从而可以抵抗缓存攻击。

多点标量乘法 在数字签名验证算法即算法 2.5 的第 9 行中，需要计算两点标量乘法之和 $s'G + tP_A$。该运算可通过合并计算两点的标量乘来进行加速，该方法称为多点标量乘，详情见文献[42]中的算法 3.48。其基本思路是预计算 $iG + jP_A$，$i, j \in [0, 2^{w-1} - 1]$，其中 w 为窗口宽度，然后使用窗口算法同时计算两个点的标量乘。

该方法的缺点是需要在运行时预计算 2^w 个点，而单独计算 $s'G$ 与 tP_A 则只需在运行时预计算 2^{w-1} 个点，因此多点标量乘会增加运行时的预计算量。而且本节固定点标量乘仅需要 37 个点加运算，无需倍点运算。合并计算两点的标量乘反而会因为运行时预计算量的增大而导致整体性能下降，因此该方案不适用于本节的应用场景。

3.2.4 其它优化

观察到 SM2 签名生成算法即算法 2.4 的第 9 行：

$$s = ((1 + d_A)^{-1} \cdot (k - rd_A)) \mod n_{sm2}$$

该过程消耗 2 个模 n_{sm2} 加/减法、1 个模 n_{sm2} 求逆和 2 个模 n_{sm2} 乘法运算。本工作对其进行等价变换：

$$s = (k + r) \cdot (1 + d_A)^{-1} - r \mod n_{sm2}$$

变换后的计算消耗 3 个模 n_{sm2} 加/减法、1 个模 n_{sm2} 求逆和 1 个模 n_{sm2} 乘运算，即使用一个模 n_{sm2} 加/减法替换一个模 n_{sm2} 乘法。由于模 n_{sm2} 加/减法运算开销低于模 n_{sm2} 乘法运算，因此可以进一步提升签名生成算法的运行效率。

3.2.5 安全性分析

为抵御侧信道攻击，本节的实现避免了私钥相关的内存访问和分支操作，包括有限域运算中的模加/减、模乘/平方以及模逆运算的实现都避免了使用分支操作。在模加/减运算中，使用掩码技术替换条件加/减法运算；模乘/平方运算的实现使用恒定时间的蒙哥马利模乘算法，算

算法 3.5: 安全的预计算表访问算法

输入: table 为预计算表，其中包含 N 个表项，table = {table[0], $\cdots$, table[$N-1$]}；索引 index，$0 < index < N$，表示希望得到的表项是 table[$index$]

输出: table[$index$]

```
1 item ← 0;                                                    /* 初始化 */
2 for i ← 0 to N − 1 do
3 |   mask ← (i! = index) − 1; /* 如果 i == index，那么 mask 的二进制表示为全 1；否则
  |     为全 0 */
4 |   item ← item XOR (mask AND table[i]); /* XOR 表示异或运算，AND 表示与运算
  |     */
5 end
6 return item
```

法 2.1 最后的条件减法也使用了掩码技术进行防护；对于模逆运算，本节使用恒定运行时间和固定执行流程的费马小定理求逆方案。

此外，为抵抗缓存攻击，本节对预计算表的存储和访问做了防护，使得访问每个预计算点都需要访问相同的缓存行，隐藏了缓存行的访问模式，并使用掩码技术来获取所需要的预计算点，其基本思路见算法 3.5。结合以上技术，本节的实现可有效抵抗简单能量分析和缓存攻击这两类侧信道攻击。

3.3 性能评估

本节的测试环境包括云端和移动端，云端设备为华为云租赁的鲲鹏通用计算增强型鲲鹏 920 服务器，双核 CPU，内存为 4GB，操作系统为 64 位 ARM 版 Ubuntu 18.04 Server。移动端设备为树莓派 4，其搭载了 1.5GHz 的四核 ARM Cortex-A72 处理器，内存为 8GB，操作系统为 Ubuntu 20.04 Server。

本节所用的时间测试接口与 OpenSSL 保持一致，模 p_{sm2} 与模 n_{sm2} 乘法/求逆运算测试运行了 2^{31} 次取均值，固定点/非固定点标量乘和签名验签测试运行了 2^{23} 次取均值。详细实验数据见表 3.1 和表 3.2。

对于模 p_{sm2} 与模 n_{sm2} 乘法，本节优化的蒙哥马利模乘每秒可执行次数在华为鲲鹏处理器上分别是 OpenSSL 实现的 1.3 与 1.2 倍，在树莓派 4 上分别是 OpenSSL 实现的 1.9 与 1.3 倍，模 p_{sm2} 乘法的加速比更高是因为 p_{sm2} 具有比 n_{sm2} 更好的数值特征；对于模 p_{sm2} 与模 n_{sm2} 求逆运算，本节优化的费马小定理实现每秒可执行次数在华为鲲鹏处理器上分别是 OpenSSL 实现的 2.1 倍与 1.5 倍，在树莓派 4 上分别是 OpenSSL 实现的 3.0 与 1.9 倍，模 p_{sm2} 求逆的加速比更高同样是因为 p_{sm2} 具有比 n_{sm2} 更好的数值特征，从而更容易构造高效的费马小定理实现；对于固定点与非固定点标量乘法，本节优化的基于窗口的算法每秒可执行次数在华为鲲鹏处理器上

表 3.1　华为鲲鹏 920 处理器和树莓派 4 上 SM2 核心运算每秒可执行次数对比（$k = 10^3$）

处理器	函数名	OpenSSL[68,71,76]	本节优化	加速比
华为鲲鹏 920	模 p_{sm2} 乘法	29337k	39202k	1.3 倍
	模 n_{sm2} 乘法	29337k	36665k	1.2 倍
	模 p_{sm2} 求逆	76580	160158	2.1 倍
	模 n_{sm2} 求逆	73754	113124	1.5 倍
	固定点标量乘	2554	77550	30.4 倍
	非固定点标量乘	2572	13406	5.2 倍
树莓派 4 ARM Cortex-A72	模 p_{sm2} 乘法	6086k	11434k	1.9 倍
	模 n_{sm2} 乘法	6086k	7880k	1.3 倍
	模 p_{sm2} 求逆	16931	51568	3.0 倍
	模 n_{sm2} 求逆	16647	31845	1.9 倍
	固定点标量乘	714	22032	30.9 倍
	非固定点标量乘	717	3886	5.4 倍

表 3.2　华为鲲鹏 920 处理器和树莓派 4 上 SM2 签名和验签每秒可执行次数对比

处理器	函数名	OpenSSL	本节优化	加速比
华为鲲鹏 920	签名生成	2413	21102	8.7 倍
	验证签名	2526	8822	3.5 倍
树莓派 4 ARM Cortex-A72	签名生成	675	6534	9.7 倍
	验证签名	777	2646	3.4 倍

分别是 OpenSSL 实现的 30.4 倍与 5.2 倍，在树莓派 4 上分别是 OpenSSL 实现的 30.9 与 5.4 倍，固定点标量乘能有如此性能提升得益于我们为固定点构造的庞大的预计算表。

将本节所描述的优化技术集成到 SM2 数字签名算法中后，测试表明，本节优化的 SM2 签名生成与签名验证算法的每秒可执行次数在华为鲲鹏处理器上分别是 OpenSSL 实现的 8.7 倍与 3.5 倍，在树莓派 4 上分别是 OpenSSL 实现的 9.7 倍与 3.4 倍。签名生成算法的加速比更高是因为该算法中只涉及到一个固定点标量乘，而签名验证算法中还包含一个非固定点标量乘，而固定点标量乘的性能提升远大于非固定点标量乘，因此签名生成算法的加速比略大于签名验证算法。

3.4 本章小结

本章对我国商用密码标准 SM2 数字签名算法在国产华为鲲鹏 ARM 处理器上的性能优化进行了深入研究，旨在解决 SM2 在国产 ARM 处理器上性能欠佳的问题。针对有限域运算，本章提出了一种基于蒙哥马利 CIOS 的更高效的模乘优化实现，并设计了一种速度更快的模逆算法。针对固定点和非固定点标量乘法运算，分别设计并实现了宽度为 7 和 5 的窗口算法。在 SM2 签名生成过程中，本章提出了一种更快速的 s 值计算方法。通过这些优化措施，SM2 在国产华为鲲鹏 ARM 处理器上的签名与验证性能分别是之前实现的 8.7 倍和 3.5 倍，显著提升了 SM2 的运行效率。本研究不仅有效解决了 SM2 算法在国产 ARM 服务器上的性能瓶颈问题，还为国产硬件上的商用密码算法提供了重要的技术支持。这项工作的意义在于，不仅能推动 SM2 在国产 ARM 处理器上的广泛应用，还能进一步提升我国在信息安全领域的自主创新能力和国际竞争力。

第四章　国际 ECC 标准 X/Ed25519 算法在高性能 Intel 服务器上的优化实现研究

X25519 密钥协商算法和 Ed25519 数字签名算法由于其公开的设计与优异的性能，逐渐成为目前主流的国际 ECC 标准。Intel AVX-512 指令集于 2013 年首次推出，其中 AVX-512IFMA 扩展支持 52 位整数乘法器，是 x86-64 系列处理器中最宽的 SIMD 乘法器。然而，使用 AVX-512 指令集加速 ECC 运算的研究相对较少，AVX-512IFMA 所提供的 52 位乘法器能否显著提升 ECC 运算性能尚未得到充分验证。因此，本章旨在探索如何利用 Intel AVX-512 指令集加速 X/Ed25519 算法。为充分利用 AVX-512 指令的并行计算能力，本章深入探讨了多种并行实现策略，针对 ECC 运算的不同层次提出了多种并行优化方案。对于有限域实现，本章使用 CryptoLine 工具进行了形式化验证，以确保实现的正确性和鲁棒性。本章还研究了如何将优化后的 X/Ed25519 算法集成至复杂的 TLS 软件栈及相关应用中，以将性能提升传递至 TLS 应用层。此外，AVX2/AVX-512 硬件单元的冷启动问题最多可使密码算法遭受 3.8 倍的性能降级，本章还研究了如何缓解该问题。最终测试结果表明，本章优化的 X25519 密钥生成、Ed25519 签名和 Ed25519 验签性能分别是目前已知最快实现的 2.32 倍、1.18 倍和 1.33 倍。将这些优化集成至 TLS 应用后，端到端实验显示，TLS 1.3 每秒可完成的握手次数提升了 25% 到 35%，而 DoT（DNS over TLS）服务端的峰值吞吐率提高了 24% 到 41%。本章的研究有助于缓解 TLS 应用中因握手带来的计算压力，有助于提升 TLS 应用的整体服务质量，尤其对电子商务、社交媒体等需要在短时间内处理大量客户端请求的场景具有显著效果。

4.1　引言

以蒙哥马利（Montgomery）曲线[43] 为基础的 X25519 密钥协商算法[44] 和以扭曲 Edwards 曲线[45] 为基础的 Ed25519 数字签名算法[46] 由于其公开的设计以及优异的性能，逐渐成为目前主流的国际 ECC 标准，并在很多网络协议中被广泛使用，包括但不限于 TLS[3,4]、SSH[5–8]、IPSec[9]、OpenPGP[47]、DNSCurve[48]、X3DH[49]、Double Ratchet[50] 和 MLS[51] 等协议。Curve25519 是 X/Ed25519 底层所依赖的椭圆曲线[52]。

支持 512 位运算的 Intel AVX-512 指令集及支持 52 位无符号整数乘法器的 AVX-512IFMA 扩展被广泛认为可用于加速诸如科学计算、人工智能、数据分析、加密和安全以及图形渲染等领域的计算任务。然而，AVX-512 指令集能否提升 ECC 运算的性能仍然缺乏充分的验证，因此本章将研究 X/Ed25519 在 Intel AVX-512 指令集上的优化实现。

在当今社会，人们日益重视隐私与安全，TLS（Transport Layer Security）协议作为网络通

信的保障，通过加密和认证确保数据安全，维护用户隐私和数据完整性，对于安全的在线交易和通信至关重要。IETF（Internet Engineering Task Force）于 2018 年在标准化文档 RFC 8446[4] 中发布了 TLS 1.3 标准，其中推荐 X25519 密钥协商算法与 Ed25519 数字签名算法为标准算法。TLS 中的关键步骤之一是 TLS 握手，X/Ed25519 算法是 TLS 握手中的耗时运算。

X/Ed25519 在各种处理器上的优化实现得到了较为充分的研究，包括针对 ARM 处理器的优化[22,77,78]、Intel AVX2/AVX-512 处理器的优化[25,26,79,80]。但是，这些工作均没有探索如何将优化的 ECC 实现集成至复杂的 TLS 软件栈中，因此 TLS 应用无法受益于这些工作对 ECC 的性能改进。因此本章还将研究如何在不修改 TLS 实现和 TLS 应用代码的前提下，将优化的 ECC 实现集成至 TLS 协议中并最终集成到 TLS 应用中。这将缓解各种应用场景中 TLS 握手所带来的计算负担，有助于改进 TLS 应用的服务质量。

仔细分析了大量密码优化相关的工作[25,26,79–82] 以及对应的开源代码[83–88] 后发现，他们在对密码算法进行性能测试时，通常在计时前先运行几百次密码操作从而对相关的 CPU 硬件和缓存进行 “热身”，这种测试方法被称为 “热启动测试”。Agner Fog 针对 Intel 系列处理器的研究[89] 指出，CPU 设计者出于降低功耗的考虑，当 AVX2/AVX-512 相关的硬件执行单元在大约 675 µs 没有被使用时，部分执行单元会被设置为低功耗模式，而当有相关的指令需要执行时，会有接近 14 µs（在 4 GHz 时大约消耗 56000 个 CPU 时钟周期）的 “热身” 阶段。在 “热身” 阶段，相关指令的吞吐会比峰值吞吐低 4.5 倍，这一现象被称为 “冷启动问题”。

在 “热启动测试” 中，AVX2/AVX-512 的 “热身” 阶段被跳过，计时期间 AVX2/AVX-512 相关指令可以在峰值吞吐条件下运行。我们称直接对密码运算进行计时测试，而没有 “热身” 阶段的测试方法为 “冷启动测试”，这种测试方法会受到 AVX2/AVX-512 执行单元 “冷启动问题” 的影响，从而使测到的性能指标低于 “热启动测试” 得到的性能指标。本章的研究发现，在真实的 TLS 应用场景中，密码算法通常会以 “冷启动” 条件执行，而不是大部分工作所报告的 “热启动” 条件。因此本章将系统性地研究和分析 “冷启动问题” 对 TLS 应用的影响，并研究如何缓解该问题对密码算法带来的性能降级影响。

本章贡献 本章针对 X/Ed25519 算法在 Intel AVX-512 指令集上进行了自底向上的优化，覆盖了 ECC 运算的所有层次，具体贡献包括：

- 针对有限域运算，本章设计并实现了 8×1 路并行策略，来最大化 AVX-512 指令集的并行计算能力。为了表明 8×1 路并行策略的优势，本章还以 4×2 路有限域乘法为例，分析了其与 8×1 路策略的区别，证明 8×1 路策略可以避免大量的置换指令从而提升运行效率。本章还使用形式化验证工具 CryptoLine 对有限域实现进行了形式化验证从而保证其正确性和鲁棒性。
- 针对椭圆曲线点运算，本章设计并实现了 8×1 路和 2×4 路并行策略，前者能充分地发

挥 8×1 路有限域运算的并行能力，后者则用于构造 1×2 路双点标量乘。

- 针对标量乘运算，本章基于 8×1 路椭圆曲线点运算构造了 8×1 路和 1×8 路固定点标量乘，分别用于加速 X25519-KeyGen 和 Ed25519-Sign；基于 2×4 路椭圆曲线点运算设计并实现了基于窗口算法的 1×2 路双点标量乘，用于加速 Ed25519-Verify。
- 最终，在 Intel Xeon (Ice Lake) Platinum 8369B 处理器上的测试表明，本章的 X25519-KeyGen 吞吐是目前已知最快实现的 2.32 倍，是 OpenSSL 实现的 12.01 倍；Ed25519-Sign 吞吐是目前已知最快实现的 1.18 倍，是 OpenSSL 实现的 3.79 倍；Ed25519-Verify 吞吐是目前已知最快实现的 1.33 倍，是 OpenSSL 实现的 3.33 倍。

除此以外，本章还研究了如何在不修改 TLS 协议和 TLS 应用代码的前提下，将本章优化的 X/Ed25519 实现集成至 TLS 协议并最终集成到 TLS 应用中，并且探究 AVX2/AVX-512 执行单元的冷启动问题对 TLS 应用中密码运算的影响，并研究如何缓解冷启动问题的影响。具体的贡献包括：

- 本章基于 OpenSSL ENGINE API 设计并实现了 ENG25519 引擎，其可以在不修改 OpenSSL 代码和 TLS 应用代码的前提下，将本章优化的 X/Ed25519 集成到 OpenSSL 和 TLS 应用中，从而使 TLS 应用能受益于本章对 X/Ed25519 的性能改进。
- 基于 ENG25519 引擎，本章以 DNS over TLS（DoT）场景为例，将 ENG25519 成功地集成至 DoT 服务器 `unbound` 中并设计了端到端实验，从而验证了本章所设计的 ENG25519 引擎的实用性。
- 基于上述贡献，本章在 DoT 场景中对密码算法的性能进行了测试，发现这些密码运算通常处于 “冷启动” 运行条件，并且发现密码运算的性能均遭到了不同程度的性能降级，部分密码运算的运行时间甚至高达热启动运行时间的 3.8 倍。为了缓解冷启动问题所导致的密码运算的性能降级，本章开发了一个基于启发式 “热身” 方案的辅助线程来缓解冷启动问题，并且通过端到端实验验证了这一解决方案的有效性。
- 最终，本章设计了三种端到端实验，分别是：（1）测试客户端与服务端每秒可完成的 TLS 1.3 握手次数，本章的 ENG25519 引擎配合辅助线程，相比于 OpenSSL 实现可以完成 25% 到 35% 的提升；（2）测试客户端与服务端每秒可完成的 DoT 查询次数，本章的解决方案相比于 OpenSSL 实现刷新了新的性能记录；（3）测试 DoT 服务端的峰值吞吐，相比于 OpenSSL 实现，本章的解决方案完成了 24% 到 41% 的峰值吞吐提升。

4.2 优化方案的设计与实现

本节旨在研究如何利用 Intel AVX-512 指令集加速 X25519 密钥协商算法和 Ed25519 数字签名算法。本节的优化实现覆盖了 ECC 运算的所有层，从底向上分别是有限域运算层、椭圆曲线点运算层和标量乘法运算层。如图 4.1 所示，针对最底层的有限域运算，本节设计了 8×1 路并

行策略，从而最大化 AVX-512 指令集的并行计算能力；针对椭圆曲线点运算，本节设计了 8×1 路和 2×4 路并行策略，前者能用于构造 8×1 路和 1×8 路标量乘，后者能构造 1×2 路标量乘；最终得到 8×1 路 X25519-KeyGen、1×8 路 Ed25519-Sign 和 1×2 路 Ed25519-Verify 实现。8×1 路 X25519-KeyGen 意味着每次函数调用可以得到 8 对独立的公私钥对，1×8 路 Ed25519-Sign 和 1×2 路 Ed25519-Verify 均是每次函数调用只执行 1 次相关的操作。X25519-KeyGen 采用多路实现方式的原因是其运算不依赖于通信对端的信息，并且额外产生的公私钥对可以保存起来供后续使用，可以很方便地集成到 OpenSSL 中。而 Ed25519-Sign 和 Ed25519-Verify 的运算均依赖于通讯对端的信息，不方便将它们的多路实现集成到 TLS 中，因此采用单路实现。本节还使用形式化验证工具 CryptoLine 对本节的有限域实现进行了形式化验证从而保证其正确性和鲁棒性。在 Intel Xeon (Ice Lake) Platinum 8369B 处理器上的测试表明，本节的 X25519-KeyGen 吞吐是目前已知最快实现的 2.32 倍，是 OpenSSL 实现的 12.01 倍；Ed25519-Sign 吞吐是目前已知最快实现的 1.18 倍，是 OpenSSL 实现的 3.79 倍；Ed25519-Verify 吞吐是目前已知最快实现的 1.33 倍，是 OpenSSL 实现的 3.33 倍。

4.2.1 实现平台介绍

本章的研究面向支持 AVX-512 指令集的高性能 Intel 处理器。Intel AVX-512 是一种高级矢量扩展指令集，旨在提高处理器性能和效率。AVX-512 指令集在 2013 年首次推出，第一款支持 AVX-512 的处理器为 Intel 于 2016 年推出的 Xeon Phi x200 系列处理器。所有支持 AVX-512 指令集的 CPU 均需实现 AVX-512F 核心扩展，除此以外 AVX-512 还支持其它扩展，如 AVX-512IFMA。AVX-512IFMA 指令集首次被实现在 2018 年上市的 Cannon Lake 处理器架构中。AVX-512IFMA 扩展中提供了 52 位整数乘法器，并提供了对应的乘法累加指令。值得注意的是，AVX-512IFMA 扩展中所提供的 52 位整数乘法器是 x86 系列处理器中最宽的 SIMD 乘法器，AVX2 与 AVX-512F 指令集提供的最宽乘法器为 32 位乘法器。

AVX-512 指令集提供了 32 个 512 位寄存器，相较于 AVX2 指令集的 256 位寄存器，AVX-512 提供了更高的数据吞吐量和更强大的并行计算能力。AVX-512 指令集的 512 位寄存器支持灵活的单指令多数据（SIMD）操作，比如可以将 512 位寄存器划分为 16 路 32 位操作、8 路 64 位操作或 4 路 128 位操作。AVX-512 引入了一些新概念，包括掩码操作和广播，掩码操作允许在执行指令时选择性地应用操作，提高了指令的灵活性；广播则可以将单个数据元素复制到一个向量寄存器的所有元素中，简化了向量化计算的编程模型。下面讲解一些 AVX-512 指令的案例：

- AVX-512IFMA 中的 52 位低半部分乘法累加指令：`VPMADD52LUQ zmm1, zmm2, zmm3`。该指令对 `zmm2` 和 `zmm3` 中的 64 位通道中的整数执行 52 位无符号乘法，并将 52 位乘法结果的低半部分累加到 `zmm1` 中，该指令缩写为 `VMACLO`。假设 $a = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$，

$b = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$，$c = [c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7]$，VMACLO$(c, a, b)$ 指令对 a 和 b 的每个 64 位通道执行 52 位低半部分无符号整数乘法并与 c 对应的通道执行累加，即 c_i+ = mul52$(a_i, b_i)_l$，$i = 0, \cdots, 7$，其中 mul52 表示 52 位乘法运算，mul52$(a_i, b_i)_l$ 表示 52 位乘法结果的低 52 位。

- AVX-512IFMA 中的 52 位高半部分乘法累加指令：VPMADD52HUQ zmm1，zmm2，zmm3。该指令对 zmm2 和 zmm3 中的 64 位通道中的整数执行 52 位无符号乘法，并将 52 位乘法结果的高半部分累加到 zmm1 中，该指令缩写为 VMACHI。VMACHI(c, a, b) 指令计算 c_i+ = mul52$(a_i, b_i)_h$，$i = 0, \cdots, 7$，其中 mul52 表示 52 位乘法运算，mul52$(a_i, b_i)_h$ 表示 52 位乘法结果的高 52 位。
- 掩码加法指令：VPADDQ zmm1k1，zmm2，zmm3。这条指令将 zmm2 和 zmm3 中 64 位整数分别相加，然后根据掩码寄存器 k1 的值，选择性地将结果写入 zmm1 中，如果对应的掩码位为 0，那么结果不会写入到 zmm1 中，否则写入到 zmm1 中。同理，掩码减法指令 VPSUBQ 的功能与掩码加法类似。将掩码加法和减法指令分别缩写为 VMADD 和 VMSUB，假设掩码 k 的十六进制表示为 0×11，二进制表示为 00010001，掩码加法操作 VMADD$(a, 0\text{x}11, a, b)$ 的结果为 $[a_0 + b_0, a_1, a_2, a_3, a_4 + b_4, a_5, a_6, a_7]$，这意味着第 0 个与第 4 个通道执行了加法运算，并写入到了结果寄存器中，而其余通道则保持不变。当掩码位全为 1 时等价于不使用掩码，将非掩码版本的加法和减法指令分别缩写为 VADD 和 VSUB。
- 置换指令：VPERMQ zmm1，zmm2，imm8。这条指令将 zmm2 中的两个 256 位通道分别根据 imm8 中的索引进行置换，并将结果存储到 zmm1 中，该指令缩写为 VPERM。假设 imm8 = $0 \times B1$，其二进制表示为 10110001，其 2 比特整数表示为 “2 3 0 1”，VPERM$(a, 0 \times B1)$ 的结果为 $[a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6]$。该指令也支持掩码操作，缩写为 VMZPERM，如果对应的掩码位为 0，那么结果寄存器对应的通道将被置 0。

4.2.2 并行实现的层次结构

下文使用范式 “有限域运算”→“椭圆曲线点运算”→“标量乘法运算” 来表示并行实现的层次结构。如图 4.1 所示，Ed25519-Verify 的优化实现策略是 $8 \times 1 \rightarrow 2 \times 4 \rightarrow 1 \times 2$，也可以缩写为 1×2 路 Ed25519-Verify。用于计算通讯双方共享密钥的 X25519-Derive 是一个例外，因为其核心运算非固定点标量乘常使用蒙哥马利阶梯算法进行计算，并且往往直接基于有限域运算直接构造，因此 X25519-Derive 的范式表示为 “有限域运算”→“蒙哥马利阶梯算法”。

AVX-512 指令集的 512 位寄存器可以划分为 8×64 位，即 8 个通道，每个通道容纳 64 位。这 8 个通道可用于实现 8×1，4×2，2×4 或 1×8 路有限域运算。

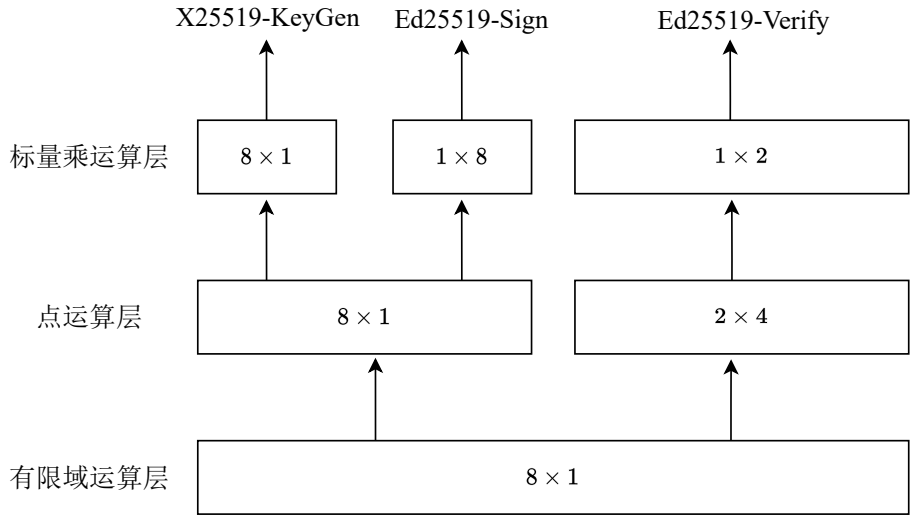


图 4.1　X/Ed25519 在 Intel 处理器上使用 AVX-512 指令集的优化实现策略概述

$m \times n$ 路并行策略　对于 $m \times n$ 路并行实现策略，“m” 表示本层并行执行的独立操作数，“n” 表示每个独立操作所使用的并行级别。以 4×2 路有限域实现为例，其中 “4” 表示并行执行 4 路独立的有限域运算，“2” 表示每一个有限域运算都利用了 2 路并行加速；图 4.1 中的 2×4 路椭圆曲线点运算，其中 “2” 表示并行执行 2 路独立的椭圆曲线点运算，“4” 表示每一路点运算内部都利用了 4 路并行加速。

并行性向上层的传递　对于 $m \times n$ 路并行实现策略，因为 “m” 表示本层并行执行的独立操作数，所以 “m” 可以传递至上一层。以图 4.1 中的 8×1 路有限域运算为例，其中的 “8” 路并行可传递至椭圆曲线点运算层，因此可以实现 8×1 路或 2×4 路点运算。同理，对于 2×4 路点运算，其中的 “2” 路并行可传递至上层的标量乘运算层，因而可以实现 1×2 路标量乘运算。

置换指令导致的性能损耗　对于 $m \times n$ 路并行实现策略，n 越大，意味着更多的并行能力被消化在了本层，也意味着需要使用更多的置换指令来调整 AVX-512 寄存器通道中的系数顺序，这些置换指令将会导致额外的性能损失。下面以 4×2 和 8×1 路有限域乘法实现为例来讲解置换指令导致的性能损耗。假设 $a, b, \cdots, g, h \in F_{p_{25519}}$，其中 $p_{25519} = 2^{255} - 19$，4×2 路有限域乘法并行计算 4 路模乘，分别是 $a \cdot e, b \cdot f, c \cdot g, d \cdot h \mod p$。本节采用 2^{51}-基数表示法来表示有限域元素，那么 a 被表示为 $\sum_{i=0}^{4} a_i 2^{51 \cdot i}$。$a$，$b$，$c$ 和 d 这四个有限域元素可用三个 512 位寄存器表示为

$$\begin{aligned} &[a_0,\ a_3,\ b_0,\ b_3,\ c_0,\ c_3,\ d_0,\ d_3] \\ &[a_1,\ a_4,\ b_1,\ b_4,\ c_1,\ c_4,\ d_1,\ d_4] \\ &[a_2,\ 0,\ b_2,\ 0,\ c_2,\ 0,\ d_2,\ 0] \end{aligned} \tag{4.1}$$

有限域元素 e，f，g 和 h 也以同样的方式进行表示。本工作利用计算序列（a_ie_j，b_if_j，c_ig_j 和 d_ih_j）来构造所需的乘积项，其中 $i,j=0,\cdots,4$：

$$\begin{aligned}
&[a_0,a_0,b_0,b_0,c_0,c_0,d_0,d_0]\times[e_0,e_3,f_0,f_3,g_0,g_3,h_0,h_3]\\
&[a_0,a_0,b_0,b_0,c_0,c_0,d_0,d_0]\times[e_1,e_4,f_1,f_4,g_1,g_4,h_1,h_4]\\
&[a_0,a_0,b_0,b_0,c_0,c_0,d_0,d_0]\times[e_2,\ 0,\ f_2,0,\ g_2,\ 0,\ h_2,0]\\
&\cdots\\
&[a_4,a_4,b_4,b_4,c_4,c_4,d_4,d_4]\times[e_0,e_3,f_0,f_3,g_0,g_3,h_0,h_3]\\
&[a_4,a_4,b_4,b_4,c_4,c_4,d_4,d_4]\times[e_1,e_4,f_1,f_4,g_1,g_4,h_1,h_4]\\
&[a_4,a_4,b_4,b_4,c_4,c_4,d_4,d_4]\times[e_2,\ 0,\ f_2,0,\ g_2,\ 0,\ h_2,0].
\end{aligned}\tag{4.2}$$

显而易见，这需要使用置换指令根据输入格式 $[a_0,a_3,b_0,b_3,c_0,c_3,d_0,d_3]$ 来获取上述所需的格式 $[a_0,a_0,b_0,b_0,c_0,c_0,d_0,d_0]$，并且累加乘积项时也需要使用置换指令。在 4×2 路有限域乘法实现的过程中，共计需要使用 20 条置换指令，其中包括 12 条`VPSHUFD`指令和 8 条`VPBLENDMQ`指令。

对于 8×1 路有限域乘法实现，上述 8 个有限域元素表示为：

$$\begin{aligned}
&[a_0,b_0,c_0,d_0,e_0,f_0,g_0,h_0]\\
&\cdots\\
&[a_4,b_4,c_4,d_4,e_4,f_4,g_4,h_4].
\end{aligned}\tag{4.3}$$

我们可以采用与单路实现相似的计算序列来构造 8×1 路有限域乘法实现，并且无需使用额外的置换指令。

综上所述，对于 $m\times n$ 路并行实现策略，我们希望尽可能地使 n 更小，减少置换指令的使用，以获得更好的性能表现。

4.2.3 8×1 路有限域运算优化实现

Cheng 等人使用 Intel AVX2 指令实现了 4×1 路 X25519 算法[26]，AVX2 支持的最宽乘法器是 32 位乘法器，因此他们采用 2^{29}-基数表示法。与 Cheng 等人的工作不同，我们采用 2^{51}-基数表示法从而能最大化利用 AVX-512IFMA 的 52 位整数乘法器的计算能力。2^{51}-基数表示法将一个 255 位的有限域元素划分为 5 个分支，有限域元素 f 被表示为：

$$f=f_0+2^{51}f_1+2^{102}f_2+2^{153}f_3+2^{204}f_4\tag{4.4}$$

其中 $0 \le f_i < 2^{52}$ 并且 $0 \le i < 5$。

在所有有限域运算中，模乘/平方运算是较为耗时且使用频繁的运算，下文将着重介绍模乘运算的实现，模平方的思路则与模乘类似。模乘 $f \times g$ 由两个步骤组成：乘法和模约减，乘法运算的计算流程为：

$$\begin{aligned} f &= f_0 + 2^{51} f_1 + 2^{51\cdot 2} f_2 + 2^{51\cdot 3} f_3 + 2^{51\cdot 4} f_4, \\ g &= g_0 + 2^{51} g_1 + 2^{51\cdot 2} g_2 + 2^{51\cdot 3} g_3 + 2^{51\cdot 4} g_4, \\ h &= f \times g = h_0 + 2^{51} h_1 + \cdots + 2^{51\cdot 8} h_8 + 2^{51\cdot 9} h_9, \\ h_k &= \sum_{i+j=k} (f_i g_j)_l + 2 \sum_{i+j=k-1} (f_i g_j)_h, \\ &\quad (0 \le i, j < 5,\ 0 \le k < 10) \end{aligned} \tag{4.5}$$

其中 $(f_i g_j)_l$ 和 $(f_i g_j)_h$ 分别表示 52 位无符号整数乘法结果的低 52 位结果和高 52 位结果。为了适配 2^{51}-基数表示法，将高 52 位结果 $(f_i g_j)_h$ 从 $2^{52} 2^{51\cdot(i+j)} (f_i g_j)_h$ 形式转换成 $2^{51\cdot(i+j+1)} 2 (f_i g_j)_h$ 形式，然后累加到对应的乘积项中。

对于模约减，因为 X25519 与 Ed25519 的模数 $p_{25519} = 2^{255} - 19$ 具有较好的数值结构，因此本节采用基于模数数值特征的快速约减算法，其计算流程为：

$$\begin{aligned} &(1) h_5 \to h_6, (2) h_6 \to h_7, (3) h_7 \to h_8, (4) h_8 \to h_9, \\ &(5) 19^2 (h_9 >> 51) \to h_0 | h_1, (6) 19 h_5 \to h_0 | h_1, \\ &(7) 19 h_6 \to h_1 | h_2, (8) 19 h_7 \to h_2 | h_3, (9) 19 h_8 \to h_3 | h_4, \\ &(10) 19 h_9 \to h_4 | h_5, (11) 19 h_5 \to h_0 \end{aligned} \tag{4.6}$$

其中，$h_5 \to h_6$ 表示将 (h_5, h_6) 替换为 $(h_5 \mod 2^{51}, h_6 + \lfloor h_5 / 2^{51} \rfloor)$；$19^2 (h_9 >> 51) \to h_0 | h_1$ 表示将 (h_0, h_1) 替换为 $(h_0 + (19^2 (h_9 >> 51))_l, h_1 + 2(19(h_9 >> 51))_h)$。在模约减过程中，$h_i (i = 5, \cdots, 9)$ 的比特长度可能会超过 52，为了后续的运算能使用 52 位乘法，需执行进位传递从而将超过 51 位的部分传递至权重更高的项中。完成进位传递后，即 (1) 到 (5)，再执行 (6) 至 (11)。上述约减操作主要利用该模等式：$2^{255} = 19 \mod p_{25519}$。

上述运算步骤中存在数据依赖，会导致额外的流水线时延，比如 (2) 中的计算需要使用 h_6，那么就必须等待 (1) 计算完成才能计算 (2)。本节将上述运算步骤的顺序调整为：(1) (3) (5) (2) (4) (6) (8) (10) (7) (9) (11)，打破了上述的数据依赖，减少了指令执行过程中流水线阻塞的情况，进而优化程序的运行效率。有限域加法和减法中的约减操作也使用相同的思路进行优化。

本节采用 Cheng 等人[26] 所使用的术语来表示 8×1 路有限域实现时的基础数据类型，即向量集 V，其由 5 个 512 位寄存器组成（v_i $(0 \le i < 5)$），每个向量包含 8 个来自不同有限域元素

```
void adc_fp255_8x1w(fe_8x1 r, const fe_8x1 a, const fe_8x1 b)
{
    v512 a0 = a[0], a1 = a[1], a2 = a[2], a3 = a[3], a4 = a[4];
    v512 b0 = b[0], b1 = b[1], b2 = b[2], b3 = b[3], b4 = b[4];
    v512 r0, r1, r2, r3, r4, t;
    const v512 mask_51b = VSET1(MASK51b), const_19 = VSET1(CONSTC);
    /** addition */
    r0 = VADD(a0, b0); r1 = VADD(a1, b1);
    r2 = VADD(a2, b2); r3 = VADD(a3, b3); r4 = VADD(a4, b4);
    /** carry r0->r1 */
    r1 = VADD(r1, VSHR(r0, 51)); r0 = VAND(r0, mask_51b);
    /** carry r2->r3 */
    r3 = VADD(r3, VSHR(r2, 51)); r2 = VAND(r2, mask_51b);
    /** carry r4->r0 */
    t = VSHR(r4, 51); t = VMUL(t, const_19);
    r0 = VADD(r0, t); r4 = VAND(r4, mask_51b);
    /** carry r1->r2 */
    r2 = VADD(r2, VSHR(r1, 51)); r1 = VAND(r1, mask_51b);
    /** carry r3->r4 */
    r4 = VADD(r4, VSHR(r3, 51)); r3 = VAND(r3, mask_51b);
    /** store result */
    r[0] = r0; r[1] = r1; r[2] = r2; r[3] = r3; r[4] = r4;
}
```

代码片段 4.1　8×1 路有限域加法实现

的分支。向量集 V 的定义为:

$$\begin{aligned} V &= [a, b, \cdots, g, h] \\ &= \left[\sum_{i=0}^{4} 2^{51i} a_i, \sum_{i=0}^{4} 2^{51i} b_i, \cdots, \sum_{i=0}^{4} 2^{51i} g_i, \sum_{i=0}^{4} 2^{51i} h_i\right] \\ &= \sum_{i=0}^{4} 2^{51i} [a_i, b_i, \cdots, g_i, h_i] = \sum_{i=0}^{4} 2^{51i} v_i \end{aligned} \tag{4.7}$$

其中 $v_i = [a_i, b_i, \cdots, g_i, h_i]$ 并且 $a_i, b_i, \cdots, g_i, h_i$ 均为 51 位或 52 位分支，每个向量由 8 个 64 位通道组成，每个通道容纳一个 51 位或 52 位分支。基于这样的数据结构，本节使用 AVX-512IFMA 指令集实现了 8×1 路有限域加法、有限域减法、有限域乘法、有限域平方和有限域求逆运算。

代码片段 4.1 展示了本节 8×1 路有限域加法实现思路，其中`fe_8x1`为上述向量集数据结

```
void mul_fp255_8x1w(fe_8x1 r, const fe_8x1 a, const fe_8x1 b)
{
    v512 h0, h1, h2, h3, h4, h5, h6, h7, h8, h9, t = VZERO;
    /** {h1,h0}+=a[0]*b[0] */
    h0 = VMACLO(VZERO, a[0], b[0]); h1 = VMACHI(VZERO, a[0], b[0]);
    h1 = VSHL(h1, 1);
    /** {h2,h1}+=a[0]*b[1]+a[1]*b[0] */
    h1 = VMACLO(h1, a[0], b[1]); h1 = VMACLO(h1, a[1], b[0]);
    t = VMACHI(t, a[0], b[1]); t = VMACHI(t, a[1], b[0]);
    h2 = VSHL(t, 1); t = VZERO;
    /** {h3, h2} += a[0]*b[2]+a[1]*b[1]+a[2]*b[0] */
    h2 = VMACLO(h2, a[0], b[2]); h2 = VMACLO(h2, a[1], b[1]);
    h2 = VMACLO(h2, a[2], b[0]); t = VMACHI(t, a[0], b[2]);
    t = VMACHI(t, a[1], b[1]); t = VMACHI(t, a[2], b[0]);
    h3 = VSHL(t, 1); t = VZERO;
    // ......
}
```

代码片段 4.2　8 × 1 路有限域乘法实现中乘法计算的部分代码示例

构，v512为 512 位数据类型，VSHR表示右移指令。

代码片段 4.2 展示了本节 8 × 1 路有限域乘法实现中乘法计算的部分代码，其中核心计算是利用 AVX-512IFMA 所提供的 52 位无符号整数乘法指令VMACLO和VMACHI来计算乘积项，其中VSHL表示左移指令。

代码片段 4.3 展示了本节 8 × 1 路有限域乘法实现中约减计算的部分代码，值得注意的是，其中第 19 行因为 h_0 的有效比特位数可能会超过 52 位，为了后续相关运算能使用 52 位乘法指令，因此将 h_0 中超出 51 位的部分进位到 h_1 中。

有限域实现的形式化验证　在形式化验证密码程序的领域中，通常使用两种方法。第一种方法采用“通过构建正确（correct-by-construction）”的方法来生成正确的实现，例如 HACL[90]、Jasmin[91] 和 Fiat[92]，而第二种方法则使用形式化验证工具来评估现有程序的正确性，例如 CryptoLine 相关工作[93]，CryptoLine 工具用于形式化验证密码原语中底层算术实现的正确性。

本节的主要目的是使用汇编语言或 Intel 内联汇编语言优化 ECC 算法的实现，然后对实现进行验证，因此本节采用 CryptoLine 工具对有限域实现进行形式化验证。本节的验证侧重于有限域实现，不包括点运算和标量乘法。主要原因是：CryptoLine 工具不支持自动或半自动地验证点运算和标量乘法的实现。其次，人工审查底层的有限域实现比点运算和标量乘法更具挑

```
void mul_fp255_8x1w(fe_8x1 r, const fe_8x1 a, const fe_8x1 b)
{
    // ......
    /** (1) carry h5->h6 */
    h6 = VADD(h6, VSHR(h5, 51)); h5 = VAND(h5, mask_51b);
    /** (3) carry h7->h8 */
    h8 = VADD(h8, VSHR(h7, 51)); h7 = VAND(h7, mask_51b);
    /** (5) carry 19^2(h9>>51)->{h1,h0} */
    t = VADD(t, VSHR(h9, 51)); h9 = VAND(h9, mask_51b);
    h0 = VMACLO(h0, t, const_361); t = VZERO;
    /** (2) carry h6->h7 */
    h7 = VADD(h7, VSHR(h6, 51)); h6 = VAND(h6, mask_51b);
    /** (4) carry h8->h9 */
    h9 = VADD(h9, VSHR(h8, 51)); h8 = VAND(h8, mask_51b);
    /** (6) carry 19*h5->{h1,h0} */
    h0 = VMACLO(h0, h5, const_19); t = VMACHI(t, h5, const_19);
    h1 = VADD(h1, VSHL(t, 1)); t = VZERO; h5 = VZERO;
    /** carry h0->h1 */
    h1 = VADD(h1, VSHR(h0, 51)); h0 = VAND(h0, mask_51b);
    // ......
}
```

代码片段 4.3　8×1 路有限域乘法实现中约减计算的部分代码示例

战性。因此，本节通过形式化验证确保了有限域实现的正确性，并通过人工审查和已知结果测试来保证点运算和标量乘法实现的正确性。

本节验证了有限域实现的计算正确性和输入输出范围正确性。使用 CryptoLine 工具进行验证的第一步是通过在GDB调试器环境中运行程序并使用`itrace.py`脚本提取执行轨迹。提取的轨迹被保存到一个后缀为`gas`的文件中。然后手动定义转换规则，并使用`to_zdsl.py`程序将`gas`文件中的汇编指令转换为 CryptoLine 语言，从而得到一个后缀为`cl`的文件。还需要手动编辑`cl`文件以改进输入输出参数的可读性，为内存变量赋值，并编写断言语句来判断所验证子程序的计算和输入输出范围的正确性。最后，通过使用`cl`文件作为输入参数运行可执行程序`cv`来确认所验证子程序的计算和输入输出范围的正确性。最终，本节的有限域实现通过了上述验证，表明本节实现的计算正确性和输入输出范围得到了保证。

4.2.4 8×1 路和 2×4 路点运算优化实现

X25519-KeyGen 与 Ed25519-Sign 中均需要计算一个固定点标量乘法，X25519-KeyGen 常复用 Ed25519-Sign 中的固定点标量乘进行计算，即在扭曲 Edwards 曲线上进行计算，然后再将计算结果转换到与之等价的蒙哥马利曲线上。用于计算通讯双方共享密钥的 X25519-Derive 常采用蒙哥马利阶梯算法即算法 2.6 进行实现，本节采用 Hisil 等人的实现[79,84]。下文将重点关注扭曲 Edwards 曲线上的椭圆曲线点运算。

对于扭曲 Edwards 曲线上的点加运算，给定两椭圆曲线点 $P=(X_P,Y_P,T_P,Z_P)$ 和 $Q=(X_Q,Y_Q,T_Q,Z_Q)$，点加运算 $R=P+Q=(X_R,Y_R,T_R,Z_R)$ 在式 2.13 中给出，式 2.13 的具体计算序列为：

$$\begin{aligned}
&A\leftarrow(Y_P-X_P)\times\left(Y_Q-X_Q\right), && B\leftarrow(Y_P+X_P)\times\left(Y_Q+X_Q\right),\\
&C\leftarrow 2d\times T_P\times T_Q, && D\leftarrow 2Z_P\times Z_Q,\\
&E\leftarrow B-A, && F\leftarrow D-C,\\
&G\leftarrow D+C, && H\leftarrow B+A,\\
&X_R\leftarrow E\times F, && Y_R\leftarrow G\times H,\\
&Z_R\leftarrow F\times G, && T_R\leftarrow E\times H.
\end{aligned} \tag{4.8}$$

对于固定点标量乘实现，在计算点加运算时两个椭圆曲线点中的一个常常从预计算表中读取，预计算表一般是非运行时构造的，不消耗运行时的计算时间。因此上式中的部分运算可在预计算阶段执行，从而提升点加运算的效率。假设上式中的点 Q 是从预计算表中读取的，其预计算格式坐标定义为：

$$Q^{pre}=(Y_Q-X_Q,Y_Q+X_Q,2dT_Q,2Z_Q) \tag{4.9}$$

以这样的方式，式 4.8 可以减少一个有限域加法、一个有限域减法、一个与常量 $2d$ 的有限域乘法和一个与常量 2 的有限域乘法。

倍点运算 $R=2P=(X_R,Y_R,T_R,Z_R)$ 在式 2.14 中给出，具体计算序列为：

$$\begin{aligned}
&A\leftarrow X_P^2, && B\leftarrow Y_P^2,\\
&C\leftarrow 2Z_P^2, && D\leftarrow -A,\\
&E\leftarrow(A+B)-(X_P+Y_P)^2, && F\leftarrow(A-B)+2Z_P^2,\\
&G\leftarrow A-B, && H\leftarrow A+B,\\
&X_R\leftarrow E\times F, && Y_R\leftarrow G\times H,\\
&Z_R\leftarrow F\times G, && T_R\leftarrow E\times H.
\end{aligned} \tag{4.10}$$

使用本节的 8×1 路有限域实现，其中每一路运算都计算一个独立的上述点加公式，因此得

算法 4.1: 扩展坐标与预计算格式坐标下扭曲 Edwards 曲线上 2×4 路点加实现

输入: 两个椭圆曲线点 P 和 J 的向量集格式 $[P,J]=[X_P,Y_P,T_P,Z_P,X_J,Y_J,T_J,Z_J]$；另外两个椭圆曲线点 Q^{pre} 和 K^{pre} 的向量集格式 $[Q^{pre},K^{pre}]=[Y_Q-X_Q,Y_Q+X_Q,2dT_Q,2Z_Q,Y_K-X_K,Y_K+X_K,2dT_K,2Z_K]$；其中 P, Q, J, 和 K 是扭曲 Edwards 曲线上的椭圆曲线点，Q^{pre} 和 K^{pre} 分别是点 Q 和 K 的预计算格式，它们通常是从预计算表中获取到的。

输出: $[R,S]=[X_R,Y_R,T_R,Z_R,X_S,Y_S,T_S,Z_S]$ 且满足 $R=P+Q$, $S=J+K$。

1 $M \leftarrow \text{VPERM}([P,J],\text{0xB1})$
2 $N \leftarrow \text{VMADD}([P,J],\text{0x11},M,2p)$
3 $N \leftarrow \text{VMSUB}(N,\text{0x11},N,[P,J])$
4 $N \leftarrow \text{VMADD}(N,\text{0x22},N,M)$
5 $U \leftarrow N\times[Q^{pre},K^{pre}]$; /* 8×1 路有限域乘法 */
6 $M \leftarrow \text{VPERM}(U,\text{0xDD})$
7 $N \leftarrow \text{VPERM}(U,\text{0x88})$
8 $M \leftarrow \text{VMADD}(M,\text{0x99},M,2p)$
9 $M \leftarrow \text{VMSUB}(M,\text{0x99},M,N)$
10 $M \leftarrow \text{VMADD}(M,\text{0x66},M,N)$
11 $N \leftarrow \text{VPERM}(M,\text{0x4B})$
12 $[R,S] \leftarrow M\times N$; /* 8×1 路有限域乘法 */
13 **return** $[R,S]$

到了 $8\times1\rightarrow8\times1$ 路的点加实现，倍点运算也以同样的方式实现。

Faz-Hernánde 等人[25] 使用 AVX2 指令集实现了 4×1 路有限域实现，并基于此实现了 1×4 路点加运算。受限于点加计算的计算流程，无法将其扩展到 1×8 路，因此本节基于式 4.8 设计了 2×4 路点加算法。

本节实现了两种形式的 2×4 路点加，即算法 4.1 和算法 4.2。算法 4.1 输入的椭圆曲线点中，P 和 J 为扩展坐标格式，Q^{pre} 和 K^{pre} 为式 4.9 所述的预计算坐标格式并且常常是通过读取预计算表而得到，该算法的使用场景为固定点标量乘。算法 4.2 输入椭圆曲线点全部为扩展坐标格式，该算法在将 2×4 路椭圆曲线点的格式转换为单路格式时用到。算法 4.3 给出了扩展坐标下 2×4 路倍点运算的实现。算法 4.1 ~ 算法 4.3 中`VPERM`、`VMZPERM`、`VMADD`、`VMSUB`等指令在本文 4.2.1 中给出了介绍。

4.2.5 8×1 路和 1×8 路固定点标量乘优化实现

本小节采用与 Faz-Hernánde 等人[25] 相似的术语来描述从预计算表中读取椭圆曲线点。对于固定点标量乘 kP，其中 P 是已知点，采用预计算技术可以大幅度降低固定点标量乘的计算开销，属于用空间换取时间的思路。假设标量 k 为 l 比特整数，$t=\lceil l/\omega\rceil$，$\omega>0$，预计算表包含 t 个

算法 4.2: 扩展坐标下扭曲 Edwards 曲线上 2×4 路点加实现

输入: 两个椭圆曲线点 P 和 J 的向量集格式 $[P,J]=[X_P,Y_P,T_P,Z_P,X_J,Y_J,T_J,Z_J]$；另外两个椭圆曲线点 Q 和 K 的向量集格式 $[Q,K]=[X_Q,Y_Q,T_Q,Z_Q,X_K,Y_K,T_K,Z_K]$；其中 P, Q, J, 和 K 是扭曲 Edwards 曲线上的椭圆曲线点。

输出: $[R,S]=[X_R,Y_R,T_R,Z_R,X_S,Y_S,T_S,Z_S]$ 且满足 $R=P+Q$, $S=J+K$。

$M \leftarrow \text{VPERM}([P,J],\text{0xB1})$
$N \leftarrow \text{VMADD}([P,J],\text{0x11},M,2p)$
$N \leftarrow \text{VMSUB}(N,\text{0x11},N,[P,J])$
$N \leftarrow \text{VMADD}(N,\text{0x22},N,M)$
$M \leftarrow \text{VPERM}([Q,K],\text{0xB1})$
$U \leftarrow \text{VMADD}([Q,K],\text{0x11},M,2p)$
$U \leftarrow \text{VMSUB}(U,\text{0x11},U,[Q,K])$
$U \leftarrow \text{VMADD}(U,\text{0x22},U,M)$
$U \leftarrow U\times 2d$; /* 8×1 路有限域乘法 */
$U \leftarrow U\times N$; /* 8×1 路有限域乘法 */
$U_1 \leftarrow \text{VPERM}(U,\text{0xDD})$
$U_2 \leftarrow \text{VPERM}(U,\text{0x88})$
$U_3 \leftarrow \text{VMADD}(U_1,\text{0x99},U_1,2p)$
$U_3 \leftarrow \text{VMSUB}(U_3,\text{0x99},U_3,U_2)$
$U_3 \leftarrow \text{VMADD}(U_3,\text{0x66},U_3,U_2)$
$U_4 \leftarrow \text{VPERM}(U_3,\text{0x4B})$
$[R,S] \leftarrow U_3\times U_4$; /* 8×1 路有限域乘法 */
return $[R,S]$

子表，每个子表包含 $2^{\omega-1}$ 个椭圆曲线点。预计算表定义为 $T_u=\{T_u(v)=2^{\omega u}vP \mid 1<v\le 2^{\omega-1}\}$，其中 $0\le u<t$，从该预计算表中读取一个椭圆曲线点表示为：

$$\phi(T_u,v)=\begin{cases} T_u(v), & \text{if } v>0 \\ -T_u(-v), & \text{if } v<0 \\ O, & \text{else .} \end{cases} \tag{4.11}$$

标量 k 被编码为一系列有符号整数片段 $(k_0,\cdots,k_{t-1})$，其中 $k=\sum_{j=0}^{t-1}2^{\omega j}k_j$ 并且 $-2^{\omega-1}\le k_j<2^{\omega-1}$，标量 k 的编码算法采用 Faz-Hernánde 等人工作中的算法 5[25]。在本节中，$\omega=4$，$t=64$，并且只预计算 u 为偶数时的子表 T_u 来降低内存占用。在读取预计算表时，本节的实现避免了使用私钥相关的索引来抵抗缓存侧信道攻击。

算法 4.3: 扩展坐标下扭曲 Edwards 曲线上 2×4 路倍点实现

输入: 两个扭曲 Edwards 曲线上的椭圆曲线点 P 和 J 的向量集格式
$[P,J]=[X_P,Y_P,T_P,Z_P,X_J,Y_J,T_J,Z_J]$

输出: $[R,S]=[X_R,Y_R,T_R,Z_R,X_S,Y_S,T_S,Z_S]$ 且满足 $R=2P,\ S=2J$。

$M \leftarrow \text{VPERM}([P,J],\text{0xF4})$
$M \leftarrow M\times[P,J]$; /* 8×1 路有限域乘法 */
$N_1 \leftarrow \text{VPERM}(M,\text{0x67})$
$N_2 \leftarrow \text{VMZPERM}(\text{0x77},M,\text{0x23})$
$N_3 \leftarrow \text{VMZPERM}(\text{0x11},M,\text{0x00})$
$N_4 \leftarrow \text{VMZPERM}(\text{0x99},M,\text{0x01})$
$U_1 \leftarrow \text{VADD}(N_1,\text{VADD}(N_2,\text{VADD}(N_3,\text{VSUB}(2p,N_4))))$
$U_2 \leftarrow \text{VPERM}(U_1,\text{0x1E})$
$[R,S] \leftarrow U_2\times U_1$; /* 8×1 路有限域乘法 */
return $[R,S]$

用于 X25519-KeyGen 的 8×1 路固定点标量乘 X25519-KeyGen 即算法 2.7 的核心操作是固定点标量乘，如上文所述，先在扭曲 Edwards 曲线上进行计算，然后将最终结果转换至等价的蒙哥马利曲线上。本节基于上文描述的 8×1 路椭圆曲线点运算来构造 8×1 路固定点标量乘，其中每一路标量乘运算的计算方法为：

$$kP=\sum_{j=0}^{31}\phi\left(T_{2j},k_{2j}\right)+2^4\sum_{j=0}^{31}\phi\left(T_{2j},k_{2j+1}\right). \tag{4.12}$$

将 8 份独立的上述计算公式分别放置到 8×1 路椭圆曲线点运算的每一路中，即可得到 $8\times1\rightarrow8\times1\rightarrow8\times1$ 路实现策略的固定点标量乘以及对应的 X25519-KeyGen 实现。

用于 Ed25519-Sign 的 1×8 路固定点标量乘 Ed25519-Sign 即算法 2.9 的核心操作是固定点标量乘。本节基于 Faz-Hernánde 等人[25] 的 1×4 路固定点标量乘推广出了 1×8 路固定点标量乘计算策略：

$$\begin{aligned}kP=&\sum_{j=0}^{7}\phi\left(T_{2j},k_{2j}\right)+2^4\sum_{j=0}^{7}\phi\left(T_{2j},k_{2j+1}\right)\\&+\sum_{j=8}^{15}\phi\left(T_{2j},k_{2j}\right)+2^4\sum_{j=8}^{15}\phi\left(T_{2j},k_{2j+1}\right)\\&+\sum_{j=16}^{23}\phi\left(T_{2j},k_{2j}\right)+2^4\sum_{j=16}^{23}\phi\left(T_{2j},k_{2j+1}\right)\\&+\sum_{j=24}^{31}\phi\left(T_{2j},k_{2j}\right)+2^4\sum_{j=24}^{31}\phi\left(T_{2j},k_{2j+1}\right),\end{aligned} \tag{4.13}$$

算法 4.4: 扩展坐标下扭曲 Edwards 曲线上双点标量乘

输入: (s, h, P, Q) 其中 s 和 h 是 l 位整数，P 和 Q 是椭圆曲线点，P 为固定点，Q 为非固定点。预计算表 tableP = $\{O, P, 2P, \cdots, 2^{w-1}P\}$，其中 w 为窗口宽度，O 是椭圆曲线加法群的单位元，因为点 P 为固定点，tableP 可以在程序运行前进行预计算。

输出: $R = sP + hQ$

运行时预计算：tableQ $\leftarrow \{O, Q, 2Q, \cdots, 2^{w-1}Q\}$

$(s_0, s_1, \cdots, s_{r-1}) \leftarrow \text{Recoding}_w(s)$; /* $r = \lceil l/w \rceil + 1$ */

$(h_0, h_1, \cdots, h_{r-1}) \leftarrow \text{Recoding}_w(h)$

$[M, N] \leftarrow [O, O]$

$[A, B] \leftarrow [\pm\text{tableP}[|s_{r-1}|], \pm\text{tableQ}[|h_{r-1}|]]$; /* 从预计算表中取出的点的符号与对应的标量片段的符号一致，即 s_{r-1} 和 h_{r-1} */

$[M, N] \leftarrow [M, N] + [A, B]$; /* 2×4 路点加 */

for $i \leftarrow r - 2$ *to* 0 **do**

  $[M, N] \leftarrow 2^w[M, N]$; /* w 次 2×4 倍点，即算法 4.2 */

  $[A, B] \leftarrow [\pm\text{tableP}[|s_i|], \pm\text{tableQ}[|h_i|]]$

  $[M, N] \leftarrow [M, N] + [A, B]$

end

$R \leftarrow M + N$

return R

其中 8 个累加操作使用本节的 8×1 路椭圆曲线点运算并行执行，即可得到 8×1 → 8×1 → 1×8 路实现策略的固定点标量乘，最终得到了 $8 \times 1 \rightarrow 8 \times 1 \rightarrow 1 \times 8$ 路 Ed25519-Sign 实现。

4.2.6 2×4 路双点标量乘优化实现

Ed25519-Verify 即算法 2.10 的核心操作是双点标量乘，即算法 2.10 第 10 行的 $s \cdot P + h \cdot Q$，其中 s 和 h 为 l 比特整数，P 为固定点也称为已知点或基点，Q 为非固定点。

Faz-Hernánde 等人[25] 采用了 NAF 方法[94] 来计算双点标量乘，其使用 AVX2 指令集实现的并行策略为 $4 \times 1 \rightarrow 1 \times 4 \rightarrow 1 \times 1$，这意味着所有并行性都被椭圆曲线点运算消耗了。受限于 NAF 方法固有的顺序计算模式，无法基于 NAF 方法推导更高效的并行实现策略。

本节使用基于窗口的算法来设计更高效的双点标量乘算法，窗口算法见算法 2.3。下面通过简单的例子来讲解窗口算法的基本思路。使用宽度 $w = 3$ 的窗口算法计算标量乘 $51P$ 的流程为：（1）将标量 51 编码为多个 w 位的有符号片段，即 $(1, -2, 3)$，其中每个片段处于范围 $[-2^{3-1}, 2^{3-1})$ 中并且满足 $1 \cdot 2^{3\cdot 2} - 2 \cdot 2^{3\cdot 1} + 3 \cdot 2^{3\cdot 0} = 51$。（2）从左向右扫描标量片段并执行累加操作 $R \leftarrow O, R \leftarrow R + 1P, R \leftarrow 2^3R, R \leftarrow R + (-2P), R \leftarrow 2^3R, R \leftarrow R + 3P$，其中 O 是椭圆曲线加法群的单位元，“+” 表示点加，2^3R 通过三次倍点运算得到，$1P$、$2P$ 和 $3P$ 通过读取预计算

表 4.1　Ed25519 不同窗口宽度下双点标量乘（即算法 4.4）消耗的点加（PA）和倍点（PD）运算数量。在本节的测试环境下，2×4 点加和倍点实现分别消耗 181 和 176 个 CPU 周期，因此将 0.97 个点加运算视为 1 个倍点运算

w	PD	PA	PD+0.97PA
3	257	88	342
4	259	68	325
5	260	57	315
6	267	53	318
7	276	55	329

表得到，$-2P$ 通过对 $2P$ 取负得到。

为了基于窗口算法设计双点标量乘，我们不再单独计算 $R \leftarrow R+iP$，而是并行计算 $R \leftarrow R+iP$ 和 $R' \leftarrow R'+jQ$，从而能利用本节的 2×4 路椭圆曲线点运算，其中 i 和 j 为标量片段。用这样的方法，最终得到了 $8\times1 \rightarrow 2\times4 \rightarrow 1\times2$ 路实现策略的双点标量乘，见算法 4.4，其中 $\text{Recoding}_w(s)$ 用于将标量编码为一系列的有符号片段，并且满足 $s=\sum_{i=0}^{r-1}2^{wi}s_i$，其中 $r=\lceil l/w\rceil+1$，w 为窗口宽度，$-2^{w-1}\le s_i<2^{w-1}$。$\text{Recoding}_w()$ 见 Faz-Hernánde 等人工作中的算法 5[25]。

窗口宽度 w 的选择　窗口宽度 w 的值越大，tableQ 在运行时的预计算开销就越大，算法 4.4 主循环中点加（PA）和倍点（PD）次数就会越少。算法 4.4 的计算开销由 tableQ 的预计算开销和主循环开销两部分组成。预计算开销为 $(2^{w-3}+1)PD+2^{w-3}PA$，主循环开销为 $w(r-1)PD+rPA$。表 4.1 总结了不同窗口宽度下算法 4.4 消耗的点加和倍点运算数量，其中在本节的测试环境下，2×4 点加和倍点实现分别消耗 181 和 176 个 CPU 始终周期，因此将 0.97 个点加运算视为 1 个倍点运算，从而对算法 4.4 的计算开销进行量化统计。根据表 4.1 可知，在窗口宽度 $w=5$ 时，算法 4.4 的性能最佳。

4.3　优化的 X/Ed25519 实现向 TLS 协议的集成

TLS 协议用于在网络上安全地传输数据，其主要目标是确保通信的安全性，防止信息被窃听、篡改或伪造。TLS 的工作原理包括协商加密算法、建立安全连接、进行身份验证和传输数据等步骤。以 OpenSSL 为代表的 TLS 软件栈的工程复杂性较高，使得优化的密码算法难以向 TLS 软件栈集成；除此以外，AVX2 和 AVX-512 硬件单元的冷启动问题最多可使密码算法遭受 3.8 倍的性能降级。为解决上述问题，本节以 TLS 1.3 为研究对象，基于 OpenSSL ENGINE API 设计并实现了 ENG25519 引擎，其能够在不修改 OpenSSL 代码和 TLS 应用代码的前提下将本章优

化的 X/Ed25519 实现集成至 TLS 应用中，并以 DoT（DNS over TLS）场景为例验证了所设计的 ENG25519 引擎的实用性。本节还设计了一个基于启发式的辅助线程来缓解 AVX2 和 AVX-512 硬件单元的冷启动问题，并通过端到端实验来验证这一解决方案的有效性。最终的实验测试表明，TLS 1.3 每秒可完成的握手次数提升了 25% 到 35%，DoT 服务端的峰值吞吐率提高了 24% 到 41%。

4.3.1 TLS 协议

SSL（Secure Sockets Layer）最早由 Netscape 公司在 1994 年发布，是 TLS 的前身。Netscape 分别在 1995 年和 1996 年发布了 SSL 2.0 版本和 SSL 3.0 版本，SSL 3.0 成为了一个较为安全和稳定的安全传输协议，被广泛应用于互联网通信中。TLS 1.0[95] 在 1999 年被设计出来，以取代 SSL 3.0。TLS 1.0 基本与 SSL 3.0 兼容，但修复了 SSL 3.0 中的一些安全漏洞，并增加了一些新的加密算法和安全特性。在 TLS 1.0 发布后，随着安全技术的不断发展，TLS 1.1[96] 和 TLS 1.2[3] 分别在 2006 年和 2008 年发布，引入了一些新的安全特性和加密算法，并修复了一些安全漏洞。TLS 1.2 成为了一个更安全和更强大的安全传输协议，被广泛应用于互联网通信中。TLS 1.3[4] 是目前最新的 TLS 版本，于 2018 年发布。TLS 1.3 带来了许多重要的改进，包括更快的握手过程、更强的加密算法、更好的安全性和隐私保护等。TLS 1.3 已经被广泛采用，并成为了互联网通信的主流安全传输协议。

TLS 协议有许多不同的实现，包括但不限于以下几种：

- OpenSSL[60]：目前使用最广泛的 TLS 库之一，被广泛应用于各种网络应用和操作系统中，本节便基于 OpenSSL 展开研究。
- GnuTLS[97]：GNU 项目的一部分，被许多 Linux 发行版和应用程序所采用。
- MbedTLS[98]：一个轻量级的开源加密库，适用于嵌入式设备和物联网应用。
- BoringSSL[99]：由 Google 开发的一个开源加密库，是 OpenSSL 的一个分支，专注于提供更安全和更简化的 TLS/SSL 实现。
- NSS[100]：由 Mozilla 开发的一个开源的加密库，提供了 TLS 和 SSL 协议的实现，被 Mozilla Firefox 等项目所采用。
- wolfSSL[101]：专注于提供高性能和低资源消耗的加密解决方案。
- LibreSSL[102]：是 OpenBSD 项目的一个分支，是 OpenSSL 的一个轻量级、安全的替代品，目前已被 FreeBSD、Nginx 等项目所采用。

图 4.2 给出了 TLS 1.3 握手的简化版流程图，下文对 TLS 1.3 握手进行介绍。握手的主要目的是协商 TLS 连接的安全参数，握手过程中的较为重要的两条消息是`ClientHello`和`ServerHello`，其中包含一些安全参数和一些扩展。`ClientHello`与`ServerHello`主要包含的信息为：

- 协议版本`legacy_version`，用于表明客户端或服务端所支持的 TLS 协议版本。

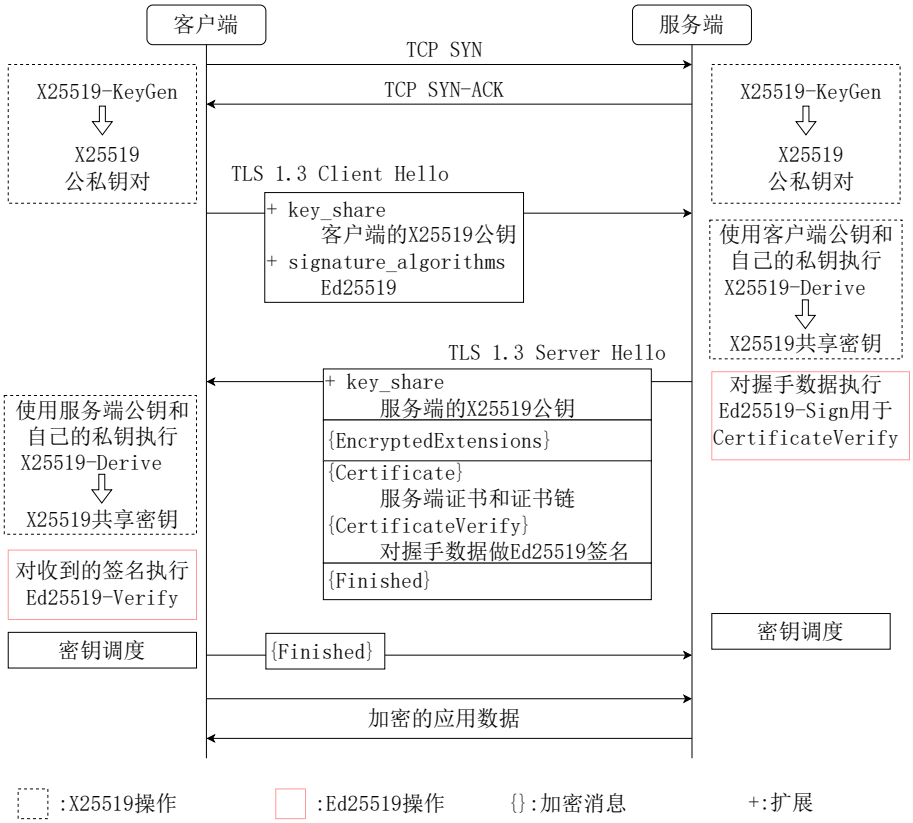


图 4.2　TLS 1.3 握手概述，使用 X25519 进行密钥交换，使用 Ed25519 进行数字签名

- 32 字节的随机数`random`，该随机数的引入可增加通信的安全性，可以确保每次握手的随机性，还能抵御重放攻击。
- 支持的密码套件列表`cipher_suites`，用于表明客户端或服务端所支持的 AEAD（Authenticated Encryption with Associated Data）算法和 HKDF（HMAC-based Extract-and-Expand Key Derivation Function）算法列表。
- `supported_groups`扩展，用于表明客户端或服务端所支持的椭圆曲线群，比如 X25519。
- `key_share`扩展，其中包含所支持的椭圆曲线密码对应的公钥，如客户端或服务端的 X25519 公钥。
- `signature_algorithms`扩展，用于表明客户端或服务端所支持的签名算法，比如 Ed25519。
- 其他扩展，比如`supported_versions`、`pre_shared_key`等。

TLS 1.3 握手中的密钥协商　假设客户端发送的`ClientHello`消息的`supported_groups`扩展中包含 X25519 算法，并且对应的`key_share`扩展中包含客户端的 X25519 公钥。服务端在收到客户端的`ClientHello`消息后，如果服务端也同样支持 X25519 算法，那么服务端即可根据客户端的 X25519 公钥和自己的 X25519 私钥通过 X25519-Derive 计算出双方的共享密钥。上述过程中

用到的 X25519 公钥和私钥均通过 X25519-KeyGen 生成。

ServerHello 中的加密扩展 服务端计算出双方的共享密钥后，进而可以通过密钥调度（key schedule）推导出用于加密通信数据的密钥，从而将`ServerHello`中的部分扩展加密后传输，并且服务端后续的消息均可加密后再发送。

TLS 1.3 握手中的认证消息 在通过上述`ClientHello`和`ServerHello`这两条消息完成密钥交换后，服务端还会发送一些用于认证的消息从而来保证身份认证、密钥确认和握手完整性。与认证相关的消息主要包含三条：

- `Certificate`消息，其中包括服务端的证书和证书链。
- `CertificateVerify`消息，服务端使用其证书对应的私钥对目前握手所产生数据的数字签名，客户端收到后可使用服务端证书公钥验证该签名，从而对服务端的身份进行验证。
- `Finished`消息，服务端对目前握手所产生数据的消息验证码（Message Authentication Code，MAC），计算 MAC 所使用的密钥是通过密钥调度步骤推导出的。客户端收到后对消息进行验证，从而保证握手的完整性和正确性。

在上述握手过程中，客户端所执行的 X/Ed25519 相关的运算包括：X25519-KeyGen、X25519-Derive 和 Ed25519-Verify；服务端执行的 X/Ed25519 相关的运算包括：X25519-KeyGen、X25519-Derive 和 Ed25519-Sign。

4.3.2 OpenSSL 与 OpenSSL ENGINE API

下文对 OpenSSL 密码库以及 OpenSSL ENGINE API 进行介绍。OpenSSL 是一个免费的开源密码库，包含了主要的密码学算法、常用的密钥和证书封装管理以及 SSL/TLS 协议。它被广泛用于各种应用程序和操作系统中，为安全通信和数据保护提供基础。OpenSSL 的主要功能包括：提供各种加密算法，例如对称加密算法（DES、AES）、非对称加密算法（RSA、ECC）、哈希函数（SHA、MD5）等。支持 SSL/TLS 协议，用于在网络通信中建立安全通道。提供密钥和证书管理功能，用于生成、存储、管理和使用密钥和证书。提供数字签名和验证功能，用于确保数据的完整性和真实性。

OpenSSL 目前已被广泛应用于各种领域，包括：服务器端应用程序，例如 Web 服务器、邮件服务器、FTP 服务器等。客户端应用程序，例如 Web 浏览器、电子邮件客户端、文件传输客户端等。操作系统，例如 Linux、FreeBSD、NetBSD、OpenBSD 等。

OpenSSL ENGINE API[103] 是一种扩展机制，用于将第三方提供的加密库或硬件设备集成到 OpenSSL 中。通过 Engine API，OpenSSL 可以透明地使用这些外部资源，而无需修改应用程序代码。OpenSSL 实现了一个内置的 “dynamic” 引擎，其可以根据配置文件在运行时加载其他的引擎，如本章所提出的 ENG25519 引擎。因此，通过修改 OpenSSL 配置文件，dynamic 引擎

表 4.2　ENG25519 引擎的详细配置

子程序	实现策略
X25519-KeyGen Ed25519-KeyGen	$8\times1 \rightarrow 8\times1 \rightarrow 8\times1$ 路并行策略 *batch-size* = 16（本文的 4.2.5 小节）
X25519-Derive	Hisil 等人[79] 的 $4\times2 \rightarrow 1\times4$ 路并行策略
Ed25519-Sign	$8\times1 \rightarrow 8\times1 \rightarrow 1\times8$ 路并行策略 （本文的 4.2.5 小节）
Ed25519-Verify	$8\times1 \rightarrow 2\times4 \rightarrow 1\times2$ 路并行策略 （本文的 4.2.6 小节）

可以在运行时以动态链接库的形式加载本节的 ENG25519 引擎，从而使本章优化的 X/Ed25519 实现透明地集成到 OpenSSL 和 TLS 应用中。

实现一个 OpenSSL 引擎需着重关注的两个功能为：

- 通过实现`bind()`函数将该引擎的相关信息告知 OpenSSL 内部的 dynamic 引擎。当 OpenSSL 内部的 dynamic 引擎加载该引擎时，首先调用该引擎所实现的`bind()`函数，该函数用于设置引擎的内部状态、分配所需的资源、并绑定与该引擎生命周期相关的函数，包括初始化函数`init()`、该引擎使用完之后调用的函数`finish()`和注销该引擎时调用的函数`destroy()`。
- 密码算法的实现。在引擎中实现各类密码算法主要通过 OpenSSL 所提供的各类数据结构来实现，比如`EVP_MD`结构用于实现哈希算法也称为消息摘要算法、`EVP_PKEY_meth`结构用于实现公钥密码算法，如本章所研究的 X25519 密钥协商算法和 Ed25519 数字签名算法、`EVP_PKEY_ASN1_meth`结构用于实现公私钥的编解码功能。

4.3.3　本工作的 ENG25519 引擎

本节采用文献[103] 中的基本方法来实现 ENG25519 引擎，从而将本章优化的 X/Ed25519 实现集成到 OpenSSL 中。ENG25519 引擎基于文献[103] 中的`libsuola`引擎[104] 和文献[105] 中的`engntru`引擎[106] 进行开发。值得注意的是，虽然`libsuola`引擎具备将优化实现的 X/Ed25519 集成到 OpenSSL 中的能力，但却因为其存在两个 bug 导致无法将其集成到 TLS 应用中。

在 ENG25519 引擎中，X25519 和 Ed25519 是通过 OpenSSL 所提供的`EVP_PKEY_meth`和对应的`EVP_PKEY_ASN1_meth`结构实现的。使用`EVP_PKEY_meth`实现 X25519 算法时，主要涉及到`keygen()`和`derive()`接口。使用`EVP_PKEY_ASN1_meth`实现 Ed25519 算法时，主要涉及`keygen()`、`sign()`和`verify()`接口。ENG25519 引擎的详细配置见表 4.2，其中 X25519-KeyGen 和 Ed25519-

KeyGen 均采用本文 4.2.5 中的 $8\times1 \rightarrow 8\times1 \rightarrow 8\times1$ 路并行策略，X25519-Derive 采用 Hisil 等人[79] 的 $4\times2 \rightarrow 1\times4$ 路并行策略，Ed25519-Sign 采用本文 4.2.5 中的 $8\times1 \rightarrow 8\times1 \rightarrow 1\times8$ 路并行策略，Ed25519-Verify 采用本文 4.2.6 中的 $8\times1 \rightarrow 2\times4 \rightarrow 1\times2$ 路并行策略。

ENG25519 引擎中 X25519 的实现 本文 4.2.5 所实现的 8×1 路 X25519-KeyGen 与常见的单路实现不同，常见的单路实现一次只生成一个公私钥对，而本文的 8×1 路实现一次可生成 8 个公私钥对，因此 ENG25519 引擎需要对公私钥对进行管理。在 ENG25519 引擎中，本节设计了`BATCH_STORE`结构来管理公私钥对。X25519 算法所对应的`keygen()`方法会访问该结构来判断其中是否还有剩余的公私钥对，如果有剩余，那么直接从该结构中获取即可；如果没有剩余，那么调用 8×1 路 X25519-KeyGen 函数 16 次，即批量大小 $batch-size = 16$，来向该结构中填充公私钥对，关于批量大小 $batch-size = 16$ 的决定将在下文进行讨论。X25519 所对应的`derive()`方法主要通过调用 Hisil 等人[79] 的 1×4 路 X25519-Derive 实现。

ENG25519 引擎中 Ed25519 的实现 Ed25519 算法对应的`keygen()`方法的实现思路与上述 X25519 的`keygen()`实现相同，也利用`BATCH_STORE`结构来管理公私钥对。`sign()`和`verify()`方法分别通过调用本文 4.2.5 的 1×8 路 Ed25519-Sign 和本文 4.2.6 的 1×2 路 Ed25519-Verify 来实现。

基于本节所述的 ENG25519 引擎，通过修改 OpenSSL 配置文件，即可将本章优化的 X/Ed25519 算法集成至 OpenSSL，并能进一步集成至 TLS 应用中。本章选择 DoT 服务器`unbound`[107] 作为 TLS 应用案例，将 ENG25519 引擎集成至`unbound`中仅仅需要修改 OpenSSL 的配置文件，无需修改 OpenSSL 和`unbound`的源码。

4.3.4 冷启动问题的缓解

正如本章 4.1 节所述，大量已知的密码优化相关的工作[25,26,79–82]，包括本文表 4.5 和表 4.6 中的测试均使用 “热启动测试” 方法来进行性能测试，这种测试方法可以避开 AVX2/AVX-512 执行单元的 “冷启动问题”[89,108,109]。而且目前已知的密码工程领域的相关工作中，对冷启动问题的探索甚少。因此，本节研究在 TLS 应用场景中冷启动问题会对 AVX2/AVX-512 实现的 ECC 带来多大影响，并研究如何缓解冷启动问题所导致的负面影响。

不同于常见的 “热启动测试”，本节基于 ENG25519 引擎与 DoT 服务器`unbound`所搭建的端到端实验系统，对 X/Ed25519 在 DoT 查询这一真实应用场景中的性能进行了测试，具体的实验配置见本章 4.4。测试结果见表 4.3，对比表中 “DoT 冷启动” 列与 “热启动” 列可知，AVX2/AVX-512 实现的 X/Ed25519 算法的性能在 DoT 场景中均受到冷启动问题的影响，其中 X25519-KeyGen 在冷启动条件下的计算开销是热启动条件的 3.8 倍，其他函数也受到了不同程度的性能降级。

CPU 的缓存未命中也会导致一定程度的性能降级，比如：如果可执行文件大于 CPU 的 L1 缓存，即使不考虑 L2 和 L3 级缓存的影响，在程序执行过程中可能会出现 L1 缓存未命中。因

表 4.3　Intel Xeon Platinum 8369B 处理器上，X/Ed25519 算法在热启动和冷启动测试条件下的时钟周期对比。热启动指在对密码算法计时测试前先对 AVX2/AVX-512 执行单元进行 “热身”，数据来源于表 4.5 和表 4.6；DoT 冷启动指在 DoT 场景下对密码算法的性能测试；DoT 热启动指在 DoT 场景下使用辅助线程对 AVX2/AVX-512 执行单元进行 “热身” 时的测试

函数	热启动	DoT 冷启动	DoT 冷启动/热启动	DoT 热启动[3]	DoT 热启动/热启动
本文的 X25519-KeyGen[1]	7560[2]	28450	3.8	10315[4]	1.4
本文的 Ed25519-Sign	24723	52157	2.1	28515	1.2
本文的 Ed25519-Verify	81506	239176	2.9	90956	1.1
Hisil 等人的 X25519-Derive[79]	61867	97789	1.6	64395	1.0

[1] 此处，批量大小设为 1 从而能更直观地观察辅助线程的效果。

[2] 在表 4.5 中，8 × 1 路 X25519-KeyGen 生成 8 个公私钥对消耗 60479 周期。7560 周期表示生成 1 个公私钥对的平均开销。

[3] 在 DoT 场景中使用辅助线程来缓解冷启动问题。

[4] ENG25519 引擎中的`BATCH_STORE`结构也存在性能消耗。

此本节使用`pmu-tools`[110]工具并结合自顶向下的分析方法（top-down method[111]）针对缓存未命中对程序的性能影响进行了探索，发现缓存未命中对程序性能的影响很小，几乎可以忽略不计。

考虑到 AVX2/AVX-512 的冷启动问题是 CPU 内部的能耗管理策略所引入的，并且 CPU 设计者没有提供相关的接口来调整这一机制，因此下文将探索如何缓解这一问题。

根据 Agner Fog 的研究[89] 可知，AVX2/AVX-512 执行单元的 “热身” 阶段大约持续 14 µs（在 4 GHz 时大约消耗 56000 个 CPU 时钟周期），“热身” 完成后所有相关的执行单元都会被激活，相关的指令就能以峰值吞吐执行。当大约 675 µs 没有执行相关指令时，相关执行单元就会被设置为低功耗模式。一个理想的解决方法是在需要运行密码操作的 14 µs 前执行一条 AVX-512 指令，从而让相关硬件提前完成 “热身”。但在 TLS 场景中精准地预测何时客户端会向服务端发起一个 TLS 握手请求是很难的。

为缓解冷启动问题，本节设计了一个启发式的 “热身” 辅助线程，其会根据服务端当前的负载情况来执行不同的操作。该辅助线程的基本工作流程是：该线程每隔 500 µs 被唤醒一次[1]，根据服务端当前的负载情况来判断是否需要执行一条 AVX-512 指令来唤醒相关的硬件执行单元，然后再次进入睡眠状态等待下次被唤醒。

[1]使用 500 µs 的时间间隔可以保证即使定时器有误差，辅助线程也能正常工作

unbound服务器提供了数据统计功能[112]，这使得我们能够得知unbound每 60 秒[1]处理了多少个 DoT 查询请求，将其表示为 Q，这将是辅助线程判断服务端负载的主要依据。将 Q_{max} 记为unbound在 60 秒内最多可处理的 DoT 请求数量，在本章的实验配置下，其大约为 290000。将 DoT 服务端的负载分为三种情况进行讨论：高负载、中负载和低负载。

对于高负载场景，即 $Q \geq 120000$，平均每 500 μs 就会有一条 DoT 查询请求，这就能保证相关执行单元不会进入低功耗模式，因此辅助线程无需做任何操作。对于中负载场景，即 $60000 \leq Q < 120000$，辅助线程每 500 μs 执行一条 AVX-512 指令从而阻止相关执行单元进入低功耗模式，进而保证服务端可以高效地处理 DoT 查询请求。对于低负载场景，即 $Q < 60000$，辅助线程不执行任何操作，因为此时服务端计算资源较为充足，所以决定不去破坏 CPU 的能耗管理机制。上述启发式 “热身” 方案可以描述为：

$$T = \begin{cases} +\infty, & Q \geq 120,000 \\ 500\,\mu s, & 60,000 \leq Q < 120,000 \\ +\infty, & Q < 60,000 \end{cases} \tag{4.14}$$

其中 T 表示辅助线程执行 AVX-512 指令的时间间隔，$+\infty$ 表示辅助线程不执行任何操作。这些阈值可以根据不同的应用场景以及不同的应用需求进行调整。

表 4.3 的后两列呈现了上述辅助线程的效果，表明本节所设计的辅助线程有效地缓解了冷启动问题，使得密码操作在真实的 TLS 场景中能以更接近热启动条件下的性能运行。表中的数据是在服务端和客户端均使用了辅助线程并且 $T = 500\,\mu s$ 的设置下在unbound服务器上测试得到的。在真实的应用场景中，往往服务端对于提高峰值吞吐这一指标更为重视，并且考虑到客户端发起 DoT 请求的行为更加不可预测，因此本章除了表 4.3 中的实验给客户端配备了辅助线程外，其余实验均只给服务端配备辅助线程。在这样的配置下，客户端所执行的密码操作，包括 X25519-Derive 和 Ed25519-Verify 将不可避免地受到冷启动问题的影响，而下文即将给出的针对 X25519-KeyGen 的批量计算技术可以在一定程度上缓解冷启动问题。

关于能耗的讨论 本节所提出的辅助线程会绕过 CPU 的节省能耗策略，并且考虑到 AVX-512 指令本身能耗就较高，那么这是否意味着本章的解决方案会增加 CPU 的能耗呢？以服务端执行一次 TLS 1.3 握手所执行的密码操作为例，其中包括 X25519-KeyGen、X25519-Derive 和 Ed25519-Sign，使用本章的 ENG25519 和辅助线程的情况下，这些运算的开销共计 97.1k 周期，相比于 OpenSSL 的 275.3k 周期和 Faz-Hernandez 等人使用 AVX2 完成的 127.1k 周期，分别提升了 184% 和 31% 的性能。专业的能耗测试[113] 表明，AVX-512 的平均功耗相比于 AVX2 仅仅增加了 17%，而相比于不使用 AVX 指令的实现平均功耗甚至降低了。综上所述，本章的解决方案不仅不会增

[1]时间间隔可以通过配置文件进行调整

表 4.4　Intel Xeon Platinum 8369B 处理器上，X25519-KeyGen 不同的批量尺寸（batch size）下每组公私钥对的均摊开销，单位为 CPU 时钟周期，包含 ENG25519 引擎所引入的内存管理开销

批量尺寸	使用辅助线程时的开销	未使用辅助线程时的开销
1	10315	28450
2	9903	24977
4	9107	19388
8	9003	14108
16	8980	11406

加 CPU 能耗，而且还能通过运算性能的提升来降低 CPU 能耗。

在线密钥生成（online-KeyGen）与离线密钥生成（offline-KeyGen） 本章所研究的 TLS 握手和 DoT 应用场景中的 X25519-KeyGen 均属于在线密钥生成的范畴。离线密钥生成的场景包括：TLS/SSL 证书机构，比如服务端的证书往往是离线生成或从证书颁发机构申请得到的，而不是在 TLS 握手过程中在线生成的；加密货币场景中，通常需要专用的安全设备来保护离线生成的密钥。本章所涉及到的 X25519-KeyGen 所生成的公私钥对被安全地存储在内存中，内存安全由操作系统来保障，并不会涉及到额外的专用安全设备。ENG25519 引擎中的密钥管理也仅仅涉及到简单的内存操作，这对于 TLS 应用是透明的。

X25519-KeyGen 批量计算技术 X25519-KeyGen 的执行不依赖于通信对端的信息，这也是本章的 ENG25519 引擎可以使用 8×1 路策略的原因，即一次调用可以生成 8 组独立的公私钥对。本章的 ENG25519 引擎在生成密钥时调用 16 次 X25519-KeyGen 从而生成 128 组公私钥对，即 $batch-size = 16$，因为连续多次调用既可以有效地均摊冷启动问题所导致的额外开销，也有助于 CPU 的缓存友好性。

表 4.4 给出了不同批量尺寸下，每组公私钥对的均摊开销，相比于 $batch-size = 1$，$batch-size = 16$ 时对于使用与不使用辅助线程的情况分别提升了 13% 和 60% 的性能。进一步加大批量尺寸的效果不再明显，因此决定将批量尺寸设为 16。CPU 时钟频率为 4 GHz 时，生成一个批次的公私钥对大约消耗 0.22 至 0.33 微秒，并不会导致用户可感知的延迟。

4.4　性能评估

4.4.1　密码算法的性能评估

测试环境 本节所用于测试的机器是从阿里云租赁的 Intel Xeon (Ice Lake) Platinum 8369B，配备 8 vCPUs 和 16 GB 内存，具体的实例型号是计算优化型 c7（ecs.c7.2xlarge），其中 8 vCPUs 可理解为 4 核 8 线程。所使用的 GCC 版本是 9.2.0，本节的测试并没有关闭 Turbo Boost 选项，

原因包括：我们希望尽可能地去模拟更接近真实应用场景的测试环境；测试表明，即使不关闭 Turbo Boost 选项，实验结果仍然是可复现的。

为了对比的公平性，我们将参与对比的代码均在上述测试环境中编译并运行，其中 OpenSSL 实现[114,115]，Hisil 等人的实现[79] 和 Cheng 等人的实现[26] 均使用 “-O2” 选项进行编译。值得注意的是，OpenSSL 中的 X/Ed25519 实现是与庞大且复杂的密码库耦合的，因此我们将其中的 X/Ed25519 实现提取出来并形成了一个相对较小的工程来进行测试。Hisil 等人的工程无法直接使用，因此我们进行了简单重构。Cheng 等人的实现默认使用 Clang 编译器，我们将其切换为 GCC 编译器来保证测试条件的一致性。Faz-Hernández 等人的实现[25] 和 Nath 等人的实现[80] 默认使用 “-O3” 选项，我们没有对其进行修改，因为测试表明 “-O2” 与 “-O3” 选项对性能的影响可以忽略不记。

不同 X25519 实现的对比见表 4.5，即使 OpenSSL 中的实现是表格中最慢的实现，但与其对比仍然是有必要的。其它工作虽然提供了比 OpenSSL 更快的实现，但他们均没有将优化实现集成到 TLS 协议中，所以 OpenSSL 中的实现能体现真实应用场景下 TLS 应用中密码运算的性能。本章的 X25519-KeyGen 吞吐是 OpenSSL 实现的 12.01 倍，是 Faz-Hernández 等人实现的 3.49 倍，是 Cheng 等人实现的 2.32 倍。Hisil 等人实现的 X25519-Derive 是目前已知的最快实现。

不同 Ed25519 实现的对比见表 4.6，本章的 Ed25519-Sign 吞吐是 OpenSSL 实现的 3.79 倍，是 Faz-Hernández 等人实现的 1.18 倍；本章的 Ed25519-Verify 吞吐是 OpenSSL 实现的 3.33 倍，是 Faz-Hernández 等人实现的 1.33 倍。

表 4.7 提供了不同固定点标量乘实现的对比，本章的 1×8 路固定点标量乘吞吐是 OpenSSL 实现的 8.06 倍，是 Faz-Hernández 等人实现的 1.67 倍。

4.4.2 安全性分析

本章的优化实现所使用的指令均为恒定执行时间的指令，没有使用形如除法[1]、取余等非恒定执行时间的指令；本节的实现也避免了私钥数据相关的内存访问或分支指令。

本章的 X25519-KeyGen 和 Ed5519-Sign 实现中的固定点标量乘涉及到预计算表的读取，本节采用了一种安全的表读取方法，该方法会遍历整个子表，然后使用掩码技术得到所要的表项。使用这种安全的预计算表读取方法可以保证攻击者无法观测到任何有用的缓存行访问模式，因此不会暴露任何私钥相关的信息，其基本思路见算法 3.5。因此本章的实现可抵御一些简单的侧信道攻击，如缓存时间攻击（cache-timing attack）、分支预测攻击（branch prediction attack）以及通过观测缓存行访问模式的攻击。

[1]攻击者可以利用除法指令恢复出私钥信息[116]

表 4.5　Intel Xeon Platinum 8369B 处理器上，X25519 的 CPU 时钟周期对比，运行 20000 次取均值。Hisil 等人的实现和 Nath 等人的实现仅提供了 X25519-Derive 实现，没有提供 X25519-KeyGen 实现。其中，OpenSSL 实现所使用的指令集为 x86-64，Faz-Hernández 等人、Nath 等人和 Cheng 等人的实现所使用的指令集均为 AVX2，Hisil 等人的实现所使用的指令集为 AVX-512F，本节实现所使用的指令集为 AVX-512IFMA

实现	X25519-KeyGen			X25519-Derive		
	周期	加速比[1]	策略	周期	加速比[2]	策略[3]
OpenSSL[114]	90809	12.01	$1\times1\rightarrow1\times1\rightarrow1\times1$	90849	1.47	$1\times1\rightarrow1\times1$
Faz-H.[25]	26420	3.49	$4\times1\rightarrow4\times1\rightarrow1\times4$	71454	1.15	$2\times2\rightarrow1\times2$
Hisil[79]	-	-	-	61867	1.00	$4\times2\rightarrow1\times4$
Nath[80]	-	-	-	64832	1.05	$4\times1\rightarrow1\times4$
Cheng[26][4]	70164	2.32	$4\times1\rightarrow4\times1\rightarrow4\times1$	-	-	-
本工作	60479	1.00	$8\times1\rightarrow8\times1\rightarrow8\times1$	-	-	-

[1] 我们的 8×1 路 X25519-KeyGen 实现一次可生成 8 对独立的公私钥对。Cheng 等人的实现策略是 4×1 路，即一次可生成 4 对独立的公私钥对。其余的实现均是一次生成 1 对公私钥对。与 Cheng 等人的比例计算方法是 $(70164/4)/(60479/8)=2.32$，其余的计算与之类似。

[2] 本章没有提供更快的 X25519-Derive 实现，因此均与 Hisil 等人的实现进行对比。

[3] X25519-Derive 的并行策略表示为 “有限域运算” →“蒙哥马利阶梯算法”。

[4] Cheng 等人提供了 4×1 路 X25519-Derive 实现，但这种实现方式很难直接集成到 TLS 协议和 TLS 应用中，因此此处没有提供与之的对比。

4.4.3　TLS 1.3 握手

本节针对两类应用场景进行测试，分别是 TLS 1.3 握手和 DoT 查询。本节的实验使用两台相同配置的机器分别作为客户端和服务端，这两台机器通过内网进行通讯，内网通讯的带宽高达 10Gbps，OpenSSL 的版本是`1.1.1q`。

测试方法　本节的测试方法与 Bernstein 等人在 Usenix Security 2022 所发表的工作[105] 类似，在服务端使用`openssl s_server`工具[117] 运行一个 TLS 服务器，其会监听一个给定的端口。在客户端运行`tls_timer`工具[118] 去测试每秒可执行的握手次数，在`tls_timer`的主循环中，其先记录当前的时间戳，然后执行一定数量的 TLS 连接，然后再次记录当前的时间戳，从而能够计算出每秒可执行的 TLS 握手次数。对于每一次的 TLS 连接，该工具都会先执行 TLS 1.3 握手，握手

表 4.6　Intel Xeon Platinum 8369B 处理器上，Ed25519 的 CPU 时钟周期对比，运行 20000 次取均值。其中，OpenSSL 实现所使用的指令集为 x86-64，Faz-Hernández 等人的实现所使用的指令集均为 AVX2，本节实现所使用的指令集为 AVX-512IFMA

实现	Ed25519-Sign			Ed25519-Verify		
	周期	加速	策略	周期	加速	策略
OpenSSL[115]	93606	3.79	$1\times1\rightarrow1\times1\rightarrow1\times1$	271445	3.33	$1\times1\rightarrow1\times1\rightarrow1\times1$
Faz-H.[25]	29252	1.18	$4\times1\rightarrow4\times1\rightarrow1\times4$	108082	1.33	$4\times1\rightarrow1\times4\rightarrow1\times1$
本工作	24723	1.00	$8\times1\rightarrow8\times1\rightarrow8\times1$	81506	1.00	$8\times1\rightarrow2\times4\rightarrow1\times2$

表 4.7　Intel Xeon Platinum 8369B 处理器上固定点标量乘的 CPU 时钟周期对比

实现	时钟周期	加速比
OpenSSL[115]	71561	8.06
Faz-Hernández[25]	14813	1.67
本工作	8880	1.00

成功后不发送任何应用数据并直接关闭该连接。这个过程所记录的消耗时间覆盖了密码操作的计算开销、数据包在网络上传输的开销以及数据从操作系统内核空间转移到用户空间的开销。

图 4.3 是该实验得到的累积分布图，其中所有的配置均使用 Ed25519 作为 TLS 1.3 握手中的签名算法。对于密钥交换算法，图例中的 “P256” 和 “X25519” 分别指 OpenSSL 内置的 NIST-P256[18] 和 X25519 实现。对于图例中的 “ENG25519”，服务端采用本章的 ENG25519 引擎和本章所设计的辅助热身线程，客户端采用本章的 ENG25519 引擎并且没有使用辅助线程。“ALL-OpenSSL” 图例指使用 OpenSSL 内置的 X/Ed25519 实现所构造的一个 OpenSSL 引擎，当其与 “X25519” 配置进行对比时，能得知 OpenSSL ENGINE 框架对性能的影响。本章的 “ENG25519” 方案每秒可完成的 TLS 1.3 握手次数相比于 “P256” 或 “X25519” 完成了 25% 到 35% 的提升。

4.4.4　DoT 查询测试

本节采用两种方法对 DoT 查询进行测试，分别为：

- 端到端实验：与上文 TLS 1.3 握手相似，客户端在完成与服务端的 TLS 握手后，发起一个 DNS 查询请求，等待收到服务端的 DNS 响应后关闭该 TLS 连接，以这样的方式去测试每秒可完成的 DoT 查询次数。
- 峰值吞吐测试：使用大量的客户端同时向服务端发起 DoT 查询请求，从而得知服务端 60 秒内最多可处理的 DoT 请求数量。大规模的服务提供商会对峰值吞吐这一指标更感兴趣，

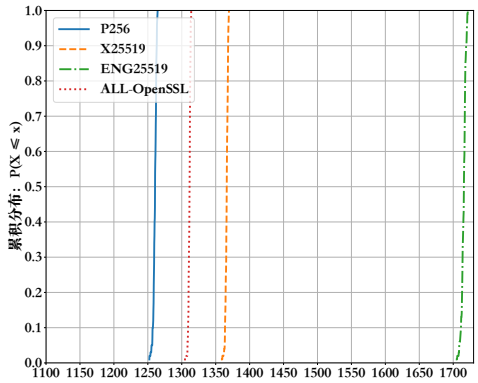


图 4.3　不同配置下每秒可完成的 TLS 1.3 握手次数，对于每种配置采样 100 次并取平均值，每次采样均执行 10000 次连续的握手

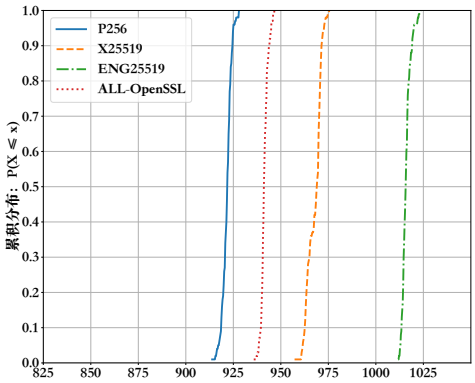


图 4.4　不同配置下每秒可完成的 DoT 查询次数，对于每种配置采样 100 次并取平均值，每次采样均执行 10000 次连续的 DoT 查询

因为提高单台机器或单个线程的吞吐可以一定程度上降低硬件采购开销。

端到端实验　服务端运行一个 DoT 服务器，即`unbound`软件[107]，其会监听特定端口上的 TLS 连接并处理客户端发来的 DoT 查询请求。客户端运行`dot_timer`，其是基于`tls_timer`所开发的用于对 DoT 查询进行测试的工具。`dot_timer`的基本工作原理与`tls_timer`相似，在完成 TLS 握手后，`dot_timer`会向`unbound`发送一个 DNS 查询请求，等待收到服务端的 DNS 响应后关闭该 TLS 连接。相比于`tls_timer`，`dot_timer`所记录的时间还额外覆盖了客户端发送 DNS 请求的时间与服务端响应 DNS 请求的时间。图 4.4 给出了该实验所得到的累计分布图，其中的图例与图 4.3 相同，本工作所提出的基于 ENG25519 引擎的解决方案优于其余配置，创下了新的性能记录。

峰值吞吐测试　大规模互联网服务提供商，如搜索引擎、云存储、电子商务和社交媒体等服务提供商，对峰值吞吐更感兴趣的原因包括：

- 更好的用户体验：对于大多数互联网服务来说，用户体验是至关重要的。如果在服务峰值期间，服务端无法处理所有请求，就会导致用户延迟、卡顿甚至服务不可用，严重影响用户体验。
- 提高资源利用率：服务端的资源通常是有限的，例如 CPU、内存和网络带宽。在大规模服务中，这些资源往往会被充分利用。如果服务端无法处理峰值流量，就会导致资源浪费。

- 提高服务可用性：在服务峰值期间，服务端更容易出现故障。如果服务端无法处理峰值流量，就有可能导致服务不可用。
- 降低成本：大规模服务提供商通常需要支付大量的基础设施费用，例如服务器租用费用和带宽费用。如果服务端的峰值吞吐量不高，就需要更多的服务器和带宽才能满足需求，从而导致成本增加。

本实验使用大量的客户端运行`dot_timer`程序同时向服务端`unbound`发起 DoT 查询请求，进而观测服务端在不同配置下的峰值吞吐。对于本工作的 ENG25519 配置，服务端峰值吞吐为每分钟 290315 次 DoT 查询，相比于 P256 配置下的每分钟 206275 次，吞吐提升了 41%；相比于 X25519 配置下的每分钟 234875 次，吞吐提升了 24%。这表明本工作的解决方案可以有效地提升服务端的吞吐，可以使大规模服务商在用户体验、资源利用率、服务可用性与降低成本等方面获得受益。

4.4.5 相关讨论

关于 AVX-512IFMA 指令集的支持 AVX-512IFMA 首次在 2018 年上市的 Cannon Lake 处理器上实现，从此以后 Intel 高性能服务器开始广泛支持 AVX-512IFMA 指令集。除了 2019 年发布的 Cascade Lake 和 2020 年发布的 Cooper Lake，2018 年以后的大部分 Intel Xeon 系列处理器均支持 AVX-512IFMA 指令集。对于客户端处理器，由于 AVX-512 指令集会增加处理器的功耗，所以客户端处理器对 AVX-512 指令集的支持略微滞后。但值得注意的是，大规模的互联网服务提供商将对更快的密码实现所带来的更高的吞吐性能更感兴趣。

本章的并行优化策略对其他处理器的启发 对于更宽的 SIMD 或 Vector 指令集，比如 ARM SVE(2) 支持的最大向量长度为 2048 比特，RISC-V 的 Vector 指令集支持的最大向量长度为 16384 比特。本章所描述的将 “利用 AVX2 的 4 路并行计算能力来加速 ECC 运算” 扩展至 “利用 AVX-512IFMA 的 8 路并行计算能力来加速 ECC 运算” 的思路，可以迁移至上述更宽的 SIMD 或 Vector 指令集上。比如，未来如果存在支持 1024 位 SIMD 或 Vector 指令集的处理器，根据本章的思路，可以考虑的并行优化策略包括：$16\times1 \to 16\times1 \to 16\times1$ X25519-KeyGen，$4\times4 \to 1\times4$ X25519-Derive，$16\times1 \to 16\times1 \to 1\times16$ Ed25519-Sign 和 $16\times1 \to 2\times8 \to 1\times2$ Ed25519-Verify。

4.5 本章小结

本章旨在探索如何利用 Intel AVX-512 指令集加速 X25519 密钥协商算法和 Ed25519 数字签名算法。本章的优化涵盖了 ECC 计算的多个方面，包括有限域运算、椭圆曲线点运算以及标量乘法运算，展示了如何通过自底向上的优化策略充分利用 AVX-512 指令集的并行性。在有限域运算的实现上，本章阐明了如何使用 AVX-512IFMA 指令集提供的 52 位乘法器来加速有限域运

算，并使用了 CryptoLine 进行形式化验证，确保实现的正确性和鲁棒性。优化后的 X/Ed25519 实现是目前已知最快的，最多是 OpenSSL 实现性能的 12 倍。这些优化方法为不同硬件平台上的 ECC 性能优化提供了有力参考。

此外，考虑到以 OpenSSL 为代表的 TLS 软件栈具有较高的工程复杂性，使得优化的密码算法难以向 TLS 软件栈集成，并且 AVX2/AVX-512 硬件单元的冷启动问题至多可使密码算法产生 3.8 倍的性能降级。为解决上述问题，本章基于 OpenSSL ENGINE API 设计并实现了 ENG25519 引擎，从而将优化后的 X25519 和 Ed25519 实现集成到 OpenSSL 和 TLS 应用中，且无需修改原有应用程序的源代码。针对 Intel AVX2/AVX-512 硬件单元的冷启动问题，本章设计并实现了一种基于启发式唤醒策略的辅助线程机制，以解决性能降级问题，该问题最多会导致密码算法性能降级 3.8 倍。最后，本章搭建了端到端实验环境，对 TLS 1.3 握手、DoT（DNS over TLS）查询以及服务端峰值吞吐量进行了性能测试。测试结果显示，本章提出的优化方案至多可使 DoT 服务器峰值吞吐量提高 41%，使大规模互联网服务提供商能够在提升用户体验、提高资源利用率、增强服务可用性以及降低运营成本等方面受益。

第五章　格密码的高效模乘算法与 NTT 优化实现研究

后量子密码（Post-Quantum Cryptography，PQC），也称抗量子密码，是一种能抵抗量子计算威胁的一类密码算法。如果大规模量子计算机得以实现，现有的 RSA 和 ECC 密码算法将不再安全。因此，后量子密码逐渐引起了学术界的广泛关注，其中格密码因其优异的安全性和性能成为最受欢迎的一类。美国国家标准与技术研究院（NIST）于 2024 年正式发布了基于模格的 ML-KEM 和 ML-DSA 标准算法。对于后量子密码标准迁移，目前学术界与工业界普遍认可的方案是将 ECC 和 PQC 混合使用；本文前两章已经对 ECC 的优化实现技术展开了研究，因此本章以及第六章将对 PQC 的优化实现展开研究。格密码中的一个耗时运算是多项式乘法，通常使用 NTT（Number Theoretic Transform）算法实现，而 NTT 算法的核心运算是整数模乘。针对格密码中多项式乘法耗时占比高的问题，本章旨在研究适用于格密码的高效模乘算法的优化实现技术，主要的研究对象包括改进版 Plantard 模乘算法和蒙哥马利模乘算法。本章先回顾 Huang 等人[119] 提出的改进版 Plantard 模乘算法以及其在 ARM Cortex-M3、ARM Cortex-M4 和单发射 RV32IM 处理器上的实现。然后本章研究了在双发射 RV32IM、双发射 RV64IM 和 RVV 处理器上，如何对改进版 Plantard 模乘和蒙哥马利模乘算法进行优化实现，并进一步研究了 ML-KEM 和 ML-DSA 的 NTT 多项式乘法的优化实现。除此以外，基于 MLWR 问题的 Saber 方案原本不支持 NTT 算法，本工作研究了如何为 Saber 适配 NTT 算法并研究了 Saber 的内存优化方法。最终的实验测试表明，本章的实现在各种处理器上均刷新了性能记录，有望促进格密码在嵌入式物联网平台上的部署和普及。

5.1　引言

格密码得益于其优异的安全性与性能在后量子密码中占据重要地位。以 ML-KEM 方案为例，其核心运算多项式乘法常使用时间复杂度为 $O(nlogn)$ 的 NTT（Number Theoretic Transform）算法进行实现，其中频繁使用的耗时运算是整数模乘，即 $a \cdot b \mod q$，其中 a 和 b 为整数，$q = 3329$。

NIST 后量子密码标准化项目在开始之初便对密码方案在嵌入式平台上的性能表现颇为重视，并且指定 ARM Cortex-M4 作为参考平台之一。ARM Cortex-M4 是一种低功耗、高性能的 ARM 处理器架构平台，广泛用于嵌入式设备和物联网（IoT）应用。格密码在各类处理器上的大部分软件优化实现研究均采用蒙哥马利模乘或 Barrett 模乘来计算整数模乘。

Alkim 等人[120] 的研究表明，得益于蒙哥马利约减算法较大的输入范围，使用延迟约减（lazy reduction）技术可以减少模约减运算的次数，进而提升 NTT 多项式乘法的性能。Seiler 等人[30]

提出了有符号版本的蒙哥马利约减算法，并给出了其在 AVX2 指令集上的实现，表明有符号版本要优于无符号版本。后续的相关工作[33,121,122] 充分研究了有符号版本的蒙哥马利约减在 ARM Cortex-M4 处理器上的实现。当前针对 16 位和 32 位模数在 Cortex-M4 处理器上最快的蒙哥马利模乘实现[121–123] 需消耗 3 个 CPU 时钟周期。

Plantard 于 2021 年提出了一种新的模乘算法[124]，称为 Plantard 模乘，原始的 Plantard 模乘并不能直接应用于格密码实现中，具体原因包括：（1）原始的 Plantard 模乘仅支持无符号运算，之前的相关工作[30,33,121–123] 表明，在实现 NTT 多项式乘法时，使用有符号运算要快于无符号运算，因为无符号运算在计算 NTT 中的蝴蝶变换时会引入额外的计算开销来处理借位逻辑。（2）原始的 Plantard 模乘的输入范围是 $[0, q]$，这意味着每次计算完一层 NTT 后均需要执行模约减来将多项式系数的范围约减到 $[0, q]$ 范围内，这些模约减运算会引入额外的性能开销。

2022 年，Huang 等人[119] 设计了适用于格密码的改进版 Plantard 模乘算法，并提供了理论证明来表明其正确性。Huang 等人还在 ARM Cortex-M3、ARM Cortex-M4 和单发射 RV32IM 处理器上实现了改进版 Plantard 模乘，以证明该算法相比于蒙哥马利模乘的优势。具体来说，改进版 Plantard 模乘相比于广泛使用的蒙哥马利模乘与 Barrett 模乘存在诸多优势：（1）所允许的输入范围更大，计算所得到的输出范围更小，因此在 NTT 实现中可使用更优的延迟约减策略来提升性能；（2）改进版 Plantard 模乘在性能上优于目前最快的蒙哥马利模乘和 Barrett 模乘。

本章贡献 针对格密码中多项式乘法耗时占比高的问题，本章研究如何在双发射 RV32IM、双发射 RV64IM 和 RVV 处理器上优化实现 ML-KEM 与 ML-DSA 算法的 NTT 多项式乘法，还研究了针对 Saber 方案的 NTT 多项式乘法适配方法以及 Saber 方案的内存优化方法，具体贡献为：

- 考虑到改进版 Plantard 模乘算法在性能上优于蒙哥马利和 Barrett 算法，但其依赖于 $l \times 2l$ 位乘法计算，因此本章先对改进版 Plantard 模乘算法的适用范围进行识别和判定。
- 对于 ML-KEM 方案的 NTT 多项式乘法实现：ML-KEM 方案的模数 $q < 2^{16}$，改进版 Plantard 模乘需使用 16×32 位乘法，因此在双发射 RV32IM 和双发射 RV64IM 上分别使用 32 位乘法器和 64 位乘法器来计算所需的 16×32 位乘法，因此可使用改进版 Plantard 模乘算法。而对于 RVV 上的实现，如果使用改进版 Plantard 模乘，那么并行性会减半，因此使用蒙哥马利模乘算法。
- 对于 ML-DSA 方案的 NTT 多项式乘法实现：ML-DSA 方案的模数 $2^{16} < q < 2^{32}$，因此在双发射 RV64IM 上可使用 64 位乘法器来计算所需的 32×64 位乘法，因此可使用改进版 Plantard 模乘算法。而对于双发射 RV32IM 和 RVV 上的实现，则使用蒙哥马利模乘算法。
- 本章还研究了 ML-KEM 与 ML-DSA 的 NTT 多项式乘法的流水线优化，本章提出的流水线调优技术能减少因乘法指令耗时较高所导致的流水线停顿，以 ML-KEM 的 NTT 在 RV32IM 上的实现为例，经过流水线优化，所消耗的时钟周期从 13218 降低到了 5714，耗

时减少了近 57%。

- 对于 Saber 方案，其原本的参数设定并不支持 NTT 多项式乘法，本工作研究了如何为 Saber 适配 NTT 算法并研究了 Saber 方案的内存优化方法，本章的实现不仅刷新了性能记录，而且还减少了 Saber 的内存占用，有助于推动 Saber 方案在内存受限的物联网设备上的部署与应用。
- 最终的性能测试表明，本章的实现均刷新了性能记录，以 ML-KEM 方案的 NTT 多项式乘法在玄铁 C908 处理器上的实现为例，本章的优化实现性能最高可达之前实现的 29.9 倍。

5.2 准备工作

本节先对所涉及到的处理器平台进行介绍，然后讲解 Huang 等人[119] 所设计的改进版 Plantard 模乘算法以及相关的优化实现。

5.2.1 实现平台介绍

本章的研究涉及多款处理器，具体包括：ARM Cortex-M3、ARM Cortex-M4、单发射 32 位 RISC-V 处理器 SiFive E31、支持 RV{32,64}GCBV 指令集的双发射玄铁 C908 处理器，下面分别进行介绍。

ARM Cortex-M3 和 ARM Cortex-M4 ARMv7-M 架构是专为嵌入式系统设计的处理器架构，广泛应用于微控制器和低功耗设备中。它支持 32 位指令集，具有精简指令集计算（RISC）的特性，提供了出色的性能与能效比。ARM Cortex-M3 和 Cortex-M4 处理器是基于 ARMv7-M 架构的典型代表。Cortex-M3 处理器定位于高性能嵌入式应用，具有低中断延迟和广泛的外设支持。Cortex-M4 则在此基础上增加了数字信号处理（DSP）指令集和浮点运算单元（FPU），进一步提升了处理器的计算能力，适用于需要更高性能的信号处理任务。

ARM Cortex-M3 与 Cortex-M4 处理器采用 3 阶段流水线设计，除了分支、乘法和内存访问指令以外的大部分指令的耗时均为 1 个时钟周期。内存加载 `ldr` 与写入 `str` 指令分别消耗 2 个和 1 个时钟周期，如果 `ldr` 指令与前一条指令存在数据依赖，那么需消耗 3 个时钟周期。当不存在指令间数据依赖时，n 条连续的 `ldr` 指令的耗时为 $n+1$ 个时钟周期，紧跟在 `ldr` 指令后面的一条 `str` 指令不消耗额外的时间。这两款处理器均支持 barrel 移位功能，支持对第二个输入操作数进行移位或旋转操作，当使用立即数指定偏移量时该功能不会引入额外的性能开销，当使用寄存器指定偏移量时该功能需额外消耗 1 个时钟周期。

Cortex-M4 的 DSP 扩展支持 SIMD 指令，一条指令可并行处理 2 路 16 位数据。Cortex-M4 上的乘法指令耗时为 1 个时钟周期，而 Cortex-M3 上的乘法指令则消耗多个时钟周期，比如乘

表 5.1　嘉楠 K230 C908 处理器常用指令的时延和 CPI（cycles per instruction）

指令	时延	CPI
RV{32,64}I		
算术/逻辑/对比指令	1	0.5
内存加载指令 `lh/lw/ld`	3	1
内存保存指令 `sh/sw/sd`	1	1
RV{32,64}M		
RV64M 上 32 位乘法指令 `mulw`	3	1
RV64M 上 64 位乘法指令 `mul{h}`	4	2
RV32M 上 32 位乘法指令 `mul{h}`	3	1
RV{32,64}B		
旋转指令 `rori` 和 `andn` 指令	1	0.5
RV{32,64}V		
向量加减法指令 `vadd/vsub`	4	1
SEW≤16 时向量乘法指令 `vmul{h}`	4	1
SEW>16 时向量乘法指令 `vmul{h}`	5	1
向量逻辑指令和 `vmerge` 指令	4	2
`vrgather` 指令	5	4
向量内存加载指令 `vle`	≥ 3	2
向量内存保存指令 `vse`	1	2.3

法累加指令 `mla` 消耗 2 个时钟周期。除此以外，Cortex-M3 上的 32 位乘法指令，包括 `umull`、`smull`、`umlal` 和 `smlal` 指令并不是恒定执行时间的，因此处理私钥相关数据时需避免使用这些指令。

RISC-V　RISC-V 是一种开源的精简指令集架构（ISA），由加州大学伯克利分校开发，旨在为各种计算平台提供灵活且可扩展的指令集标准。RISC-V 架构以模块化设计为核心，定义了一组基本的指令集，称为 RV32I（32 位整数指令集）或 RV64I（64 位整数指令集），并允许添加多个可选的扩展模块，如浮点运算、向量处理、加密扩展等。

RISC-V 的指令集采用精简指令集计算的原则，指令集简单且高效，易于实现和优化，特别适用于嵌入式系统、微控制器、高性能计算等领域。由于其开源性质，RISC-V 得到了广泛的社区支持和快速的发展，逐渐形成了丰富的硬件和软件生态系统。

SiFive E31 处理器 SiFive Freedom E310 开发板配备了 32 位单发射 E31 RISC-V 处理器[125]，支持 RV32IMAC 指令集。其配备了 16 KB 的 RAM，16 KB 的 L1 指令缓存和 16 KB 的 DTIM（Data Tightly Integrated Memory）；乘法指令（`mul` 和 `mulh`）的时延为 5 个时钟周期；`lw` 和 `sw` 指令的时延为 2 个时钟周期；`lh,lhu,lb,lu` 指令的时延为 3 个时钟周期；除法指令是非恒定执行时间的，本文的实现没有用到除法指令；除此以外，其余大部分指令的开销均为 1 个时钟周期，并且除了除法指令外，其余指令均为恒定执行时间。

玄铁 C908 处理器 嘉楠 K230 开发板[126] 配备了平头哥玄铁 C908 RISC-V 处理器[127]，主频为 1.6 GHz，配备 32 KB L1 指令缓存、32 KB L1 数据缓存、256 KB L2 缓存。C908 支持 RV{32,64}GCBV 指令集，其中向量扩展的版本为 1.0[128]，向量寄存器位长为 128（VLEN=128），B 扩展版本为 1.0.0[129]。C908 同时支持 RV64 和 RV32 两种执行模式，值得关注的特性包括：9 级流水线架构、标量指令顺序双发射、向量指令顺序单发射、支持写组合、内存读和写操作分别支持 8 路和 12 路总线并发访问。

C908 用户手册[127] 中给出了整数和乘法指令的时延，但没有提供 C908 的微架构细节，也没有提供 B 扩展和 V 扩展指令的时延。因此本节对 C908 执行了一系列基准测试，这些测试受到了文献[130,131] 的启发，测试结果见表 5.1。基本的算术、逻辑与对比指令的时延为 1，CPI（cycles per instruction）为 0.5；内存加载指令的时延为 3，内存写入指令的时延为 1，这两类指令的 CPI 均为 1。对于整数乘法指令，RV64M 上的 `mul{h}` 指令的时延与 CPI 分别为 4 和 2；RV64M 上的 `mulw` 指令与 RV32M 上的 `mul{h}` 指令均用于计算 32 位整数乘法，其时延与 CPI 分别为 3 和 1。对于 B 扩展，本章主要使用其中的旋转指令 `rori` 和 `andn` 指令，它们的时延与 CPI 分别为 1 和 0.5。对于 V 扩展，向量加减法指令和 SEW（selected element width）≤16 时的 `vmul{h}` 指令时延与 CPI 分别为 4 和 1。SEW>16 时的 `vmul{h}` 指令时延与 CPI 分别为 5 和 1。对于逻辑运算指令和 `vmerge` 指令，时延与 CPI 分别为 4 和 2。`vrgather` 指令的运行效率最低，时延与 CPI 分别为 5 和 4。对于向量内存加载 `vle` 和写入 `vse` 指令，CPI 分别为 2 和 2.3，写入指令的 CPI 略高是由写组合特性所导致的，`vle` 与 `vse` 指令的时延并不高，主要得益于多路并发总线访问特性。

除此以外，本节的测试还发现，对于指令序列 "`ld t0,(a0); add t0,t0,1`"，`add` 指令会因为等待内存加载结果而额外消耗 1 个时钟周期。为使程序达到最佳性能，应尽可能地在内存加载与计算指令间插入没有数据依赖的指令，从而避免等待内存加载结果导致的额外开销。

5.2.2 改进版 Plantard 模乘算法

算法 5.1 给出了原始的 Plantard 模乘算法[124]，其只支持无符号运算，与蒙哥马利模乘和 Barrett 模乘相同，该算法同样需要计算 3 次整数乘法。但 Plantard 模乘的优势在于：当 b 为常量

算法 5.1: 原始的 Plantard 模乘算法

输入: 两个无符号数 a, b 且满足 $a, b \in [0, q]$；$q < \frac{2^l}{\phi}$；$\phi = \frac{1+\sqrt{5}}{2}$；$q' = q^{-1} \bmod 2^{2l}$；$l$ 为机器字长，常取值为 16、32 或 64

输出: $r = ab(-2^{-2l}) \bmod q$ 并且 $r \in [0, q]$

1 $c \leftarrow abq' \bmod 2^{2l}$；　/* b 为常量时，bq' 可以被预计算 */

2 $t \leftarrow (c >> l) + 1$

3 $r \leftarrow tq >> l$

4 **return** r

算法 5.2: 改进版 Plantard 模乘

输入: 两个有符号数 a, b 且满足 $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$；$q < 2^{l-\alpha-1}$；$q' = q^{-1} \bmod^{\pm} 2^{2l}$；$q = 3329$；$l = 16$；$\alpha = 3$

输出: $r = ab(-2^{-2l}) \bmod^{\pm} q$ 并且 $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1 $c \leftarrow abq' \bmod^{\pm} 2^{2l}$；　/* b 为常量时，bq' 可以被预计算 */

2 $t \leftarrow (c >> l) + 2^{\alpha}$

3 $r \leftarrow tq >> l$

4 **return** r

时，可以通过预计算 $b \cdot (q^{-1} \bmod 2^{2l})$ 来减少一次乘法运算。但需要注意的是，$a \cdot (b(q^{-1} \bmod 2^{2l}))$ 的计算需要使用 $l \times 2l$-位乘法器。

算法 5.2 给出了 Huang 等人[119] 所设计的改进版 Plantard 模乘算法，相比于原始的 Plantard 模乘算法，改进版 Plantard 模乘支持有符号数运算并且具有更大的输入范围和更小的输出范围，因此更适合于格密码实现。下文给出算法 5.2 的计算正确性、输入输出范围的证明，并给出其与原始的 Plantard 模乘、蒙哥马利模乘与 Barrett 模乘的对比。

定理 5.1：q 为奇数，l 为使得 $q < 2^{l-\alpha-1}$ 成立的最小字长且一般为 2 的幂，其中 $\alpha > 0$，算法 5.2 对于 $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$ 是正确的。

定理 5.1 **的证明**: 算法 5.2 的正确性依赖于四个条件：

1. 输出范围的正确性：$r \in [-\frac{q+1}{2}, \frac{q}{2})$；
2. $pq - ab$ 可以被 2^{2l} 整除，其中 $p = abq^{-1} \bmod^{\pm} 2^{2l}$；
3. $0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}$，其中 $p_1 = \left\lfloor \frac{p}{2^l} \right\rfloor, p_0 = p - p_1 2^l$。因为有符号数右移操作总是将结果向负无穷方向取整，因此 $p_0 \in [0, 2^l)$；
4. 计算结果的正确性：$r = ab(-2^{-2l}) \bmod^{\pm} q$。

算法 5.2 的主要运算为：

$$r \leftarrow \left\lfloor \frac{\left(\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^{\alpha} \right) q}{2^l} \right\rfloor$$

下面依次证明上述四个条件：

1. 因为 $\left\lfloor \frac{abq' \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor \in [-2^{l-1}, 2^{l-1}-1]$，因此有：

$$\left\lfloor \frac{(-2^{l-1}+2^{\alpha})q}{2^l} \right\rfloor \leq r \leq \left\lfloor \frac{(2^{l-1}-1+2^{\alpha})q}{2^l} \right\rfloor$$
$$\left\lfloor -\frac{q}{2} + \frac{q}{2^{l-\alpha}} \right\rfloor \leq r \leq \left\lfloor \frac{q}{2} + \frac{(2^{\alpha}-1)q}{2^l} \right\rfloor$$

$\frac{q}{2^{l-\alpha}} < \frac{1}{2}$ 且 q 为奇数，因此可从上述不等式的左侧推导出 $r \geq \lfloor -\frac{q}{2} \rfloor = -\frac{q+1}{2}$。考虑到 $q < 2^{l-\alpha-1}$，因此可得：

$$\frac{(2^{\alpha}-1)q}{2^l} < \frac{q2^{\alpha}}{2^l} < \frac{2^{\alpha}2^{l-\alpha-1}}{2^l} = \frac{1}{2}$$

因为 q 为奇数，所以：

$$\left\lfloor \frac{q}{2} + \frac{(2^{\alpha}-1)q}{2^l} \right\rfloor = \left\lfloor \frac{q}{2} \right\rfloor < \frac{q}{2}$$

故 $r \in [-\frac{q+1}{2}, \frac{q}{2})$ 成立。

2. 因为 q 为奇数，故存在 $p = abq^{-1} \bmod^{\pm} 2^{2l}$ 使得 $pq - ab = 0 \bmod^{\pm} 2^{2l}$ 成立，因此 $pq - ab$ 可以被 2^{2l} 整除。

3. 先假设不等式 5.1 成立，然后根据该不等式推导 ab 的最大输入范围：

$$0 < q2^{l+\alpha} - p_0 q + ab < 2^{2l}, \tag{5.1}$$

其中 $q < 2^{l-\alpha-1}, p = abq^{-1} \bmod^{\pm} 2^{2l}, p_1 = \lfloor \frac{p}{2^l} \rfloor, p_0 = p - p_1 2^l, p_0 \in [0, 2^l)$ 并且 $\alpha > 0$。

- 当 $ab \geq 0$ 且 $0 \leq p_0 < 2^l$ 时：

$$q2^{l+\alpha} - p_0 q + ab \geq q2^{l+\alpha} - p_0 q > q2^{l+\alpha} - q2^l > 0,$$

这意味着对于 $\alpha > 0$，不等式 5.1 左侧恒成立。考虑到：

$$q2^{l+\alpha} - p_0 q + ab \leq q2^{l+\alpha} + ab < 2^{2l}$$

为了使不等式 5.1 右侧恒成立，因此 $ab < 2^{2l} - q2^{l+\alpha}$。

- 当 $ab < 0$ 且 $0 \leq p_0 < 2^l$，考虑到：

$$q2^{l+\alpha} - p_0 q + ab < q2^{l+\alpha} < 2^{l-\alpha-1}2^{l+\alpha} < 2^{2l},$$

因此对于 $\alpha > 0$，不等式 5.1 右侧恒成立。对于不等式 5.1 左侧：

$$q2^{l+\alpha} - p_0 q + ab > q2^{l+\alpha} - q2^l + ab > 0$$

为了使不等式 5.1 左侧恒成立，因此 $ab \geq q2^l - q2^{l+\alpha}$。

综上所述，对于 $ab \in [q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$，不等式 5.1 恒成立，并且可知：

$$0 < \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} < 1.$$

4. 考虑到 $pq - ab$ 可以被 2^{2l} 整除，因此：

$$\begin{aligned} ab\left(-2^{-2l}\right) &= \frac{pq - ab}{2^{2l}} = \left\lfloor \frac{pq - ab}{2^{2l}} + \frac{q2^{l+\alpha} - p_0 q + ab}{2^{2l}} \right\rfloor = \left\lfloor \frac{qp_1 2^l + q2^{l+\alpha}}{2^{2l}} \right\rfloor \\ &= \left\lfloor \frac{q(p_1 + 2^\alpha)}{2^l} \right\rfloor = \left\lfloor \frac{\left(\left\lfloor \frac{abq^{-1} \bmod^{\pm} 2^{2l}}{2^l} \right\rfloor + 2^\alpha \right) q}{2^l} \right\rfloor = r \bmod^{\pm} q. \end{aligned}$$

□

与原始的 Plantard 模乘的对比 改进版 Plantard 模乘具有诸多优势：

- 改进版 Plantard 模乘支持有符号数运算，因此更适合用于实现格密码中常用的 NTT 多项式乘法。
- 改进版 Plantard 模乘允许更大的输入范围与更小的输出范围。原始的 Plantard 模乘算法的输入范围与输出范围均为 $[0, q]$，改进的 Plantard 模乘的输入范围与输出范围分别为 $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$ 和 $[-\frac{q+1}{2}, \frac{q}{2})$，更大的输入范围与更小的输出范围允许我们在实现 NTT/INTT 时设计更佳的延迟约减策略，具体的实现细节将在下文给出。

与蒙哥马利模乘和 Barrett 模乘的对比 在格密码实现中，改进版 Plantard 模乘相比于目前使用最广泛的蒙哥马利模乘和 Barrett 模乘具有诸多优势：

- 对于改进版 Plantard 模乘，当两个输入操作数中的一个是常量时，可以通过预计算技术来节省一次乘法运算，但需要实现平台支持 $l \times 2l$ 位乘法器。因此在一些特定平台上，改进版 Plantard 模乘在性能上优于蒙哥马利模乘和 Barrett 模乘。除此以外，在计算 Barrett 模乘时通常需要执行一次额外的移位操作，这也会降低 Barrett 模乘的性能。
- 基于改进版 Plantard 模乘可以推导出对应的约减算法，其在性能上与蒙哥马利约减和 Barrett 约减相当，但在输入范围与输出范围方面存在优势。

- 改进版 Plantard 模乘的输出范围是 $[-\frac{q+1}{2}, \frac{q}{2})$，而蒙哥马利模乘的输出范围为 $[-q, q]$，再考虑到改进版 Plantard 模乘所允许的更大输入范围，因此在实现 NTT/INTT 时可以设计更佳的延迟约减策略来提升性能。

除了上述优点外，改进版 Plantard 模乘存在两个缺点：

- 实现 NTT 时，模乘的两个输入之一 b 是常量，因此可以将 $bq' \bmod^{\pm} 2^{2l}$ 进行预计算并存在预计算表中，并且计算 $a \cdot (bq') \bmod^{\pm} 2^{2l}$ 时需使用 $l \times 2l$ 位乘法器，这意味着只有实现平台能计算 $l \times 2l$ 位乘法运算时改进版 Plantard 模乘才可用，如 AVX2、ARM NEON 等指令集便不能用于实现改进版 Plantard 模乘，然而 ARM Cortex-M3、ARM Cortex-M4 与 32 位 RISC-V 处理器便可使用改进版 Plantard 模乘来加速 ML-KEM 方案。
- $bq' \bmod^{\pm} 2^{2l}$ 的预计算意味着预计算表的大小会变为原来的 2 倍，这会在一定程度上增加资源占用并增加读取预计算所需要的时间，比如计算 ML-KEM 方案的 NTT 时用到的预计算表大小会从 256 字节增加到 512 字节。

尽管如此，本章的实验结果表明，改进版 Plantard 模乘对于 ML-KEM 和 ML-DSA 方案的 NTT 多项式乘法仍然是有性能提升的。

改进版 Plantard 模乘算法的适用范围 改进版 Plantard 模乘算法依赖于 $l \times 2l$ 位乘法计算，当特定平台能有效计算 $l \times 2l$ 位乘法时，改进版 Plantard 模乘性能优于蒙哥马利模乘和 Barrett 模乘，因此下面对其适用范围进行分析。ML-KEM 与 ML-DSA 分别依赖于 16 位和 32 位 NTT 多项式乘法，我们可以用 32 位乘法器和 64 位乘法器来分别计算 16×32 位和 32×64 位乘法，因此可以使用改进版 Plantard 模乘的场景包括：RV32IM 和 RV64IM 上的 ML-KEM NTT 的实现、RV64IM 上的 ML-DSA NTT 的实现。对于 RV32IM 上的 ML-DSA NTT 实现，我们采用蒙哥马利模乘进行实现。对于 RVV（RISC-V Vector），其所支持的向量乘法指令均为 $l \times l$ 比特格式，其中 $l \in \{8, 16, 32, 64\}$，如果使用 32×32 比特乘法器来实现 16×32 比特乘法，那么相比于 16×16 比特方法并行性减半，因此 RVV 上的实现均使用蒙哥马利模乘实现。对于 RV32IM 上 Saber 的 NTT 适配，其依赖于 32 位 NTT，因此也使用蒙哥马利模乘进行实现。

5.2.3 已有实现工作

文献[119,132,133]对改进版 Plantard 模乘在 ARM Cortex-M3、ARM Cortex-M4 和单发射 RV32IM 处理器上的实现进行了研究，并介绍了如何将其应用于 ML-KEM 和 ML-DSA 方案的 NTT 多项式乘法中，下面进行具体介绍。

改进版 Plantard 模乘在 ARM Cortex-M3 处理器上的实现及其在 ML-KEM NTT 中的应用 算法 5.3 和 算法 5.4 分别给出了 Huang 等人[132]的改进版 Plantard 模乘和对应 CT 蝴蝶变换在 ARM Cortex-M3 处理器上的实现。

算法 5.3: ML-KEM NTT 中 Plantard 模乘在 Cortex-M3 上的实现

输入: $a \in [-137q, 230q]$；ζ 为常量；预计算值 $\zeta q' = \zeta q^{-1} \bmod^{\pm} 2^{2l}$

输出: $r = a\zeta(-2^{-2l}) \bmod^{\pm} q$ 并且 $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **mul** $r, a, \zeta q'$; /* $r \leftarrow a\zeta q' \bmod^{\pm} 2^{2l}$ */
2 **add** $r, 2^{\alpha}, r$, asr#16
3 **mul** r, r, q
4 **asr** r, r, #16; /* $r \leftarrow rq >> l$ */
5 **return** r

算法 5.4: ML-KEM NTT 中 CT 蝴蝶变换在 Cortex-M3 上的实现

输入: 两个有符号数 a, b；32 位旋转因子 ζ

输出: $a' = a + b\zeta, b' = a - b\zeta$

1 **mul** b, b, ζ
2 **add** $b, 2^{\alpha}, b$, asr #16
3 **mul** t, b, q
4 **sub** b', a, t, asr #16
5 **add** a', a, t, asr #16
6 **return** a', b'

算法 5.3 使用 32×32 位乘法指令 `mul` 来计算 $a\zeta q' \bmod^{\pm} 2^{2l}$。为了得到 $a\zeta q' \bmod^{\pm} 2^{2l} >> l$，可使用 1 条移位指令进行计算，但这种实现方法会额外引入 1 条移位指令。考虑到 ARM Cortex-M3 支持内联 barrel 移位操作，因此可以将 "$+2^{\alpha}$" 操作与上述移位操作结合到 1 条指令中执行，即算法 5.3 中的 `add` $r, 2^{\alpha}, r$, asr#16 指令，该指令在 ARM Cortex-M3 处理器上只消耗一个时钟周期。最终，算法 5.3 消耗 4 条指令，其中包括 2 条乘法指令、1 条移位指令和 1 条加法指令，而之前已知的最快蒙哥马利模乘实现至少需要使用 5 条指令[134]。

算法 5.4 中旋转因子的预计算格式为 32 位整数 $\zeta \leftarrow ((\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1}) \bmod^{\pm} 2^{2l}$。GS 蝴蝶变换的实现与 CT 类似，但 GS 蝴蝶变换要比 CT 蝴蝶变换多消耗 1 条指令。值得注意的是，通过利用 ARM Cortex-M3 所提供的内联 barrel 移位操作，算法 5.3 中最后一条移位指令可以与蝴蝶变换中的加减法结合起来执行，因此可以节省 1 条指令。整体来看，使用改进版 Plantard 模乘实现的 CT/GS 蝴蝶变换比蒙哥马利模乘实现版本要少使用 1 条指令[134]。考虑到 CT 蝴蝶变换可以比 GS 蝴蝶变换少使用 1 条指令，因此通常使用 CT 蝴蝶变换来实现 INTT。

除此以外，Huang 等人[132] 采用了 Abdulrahman 等人[123,135] 所提出的轻量化蝴蝶变换（light butterfly unit）技术；NTT 和 INTT 的层合并策略分别为 3+3+1 和 3+1+3。Huang 等人的实现提供了两个版本，分别是内存优化和性能优化版本，并分别为不同版本设计了不同的延迟约减策略。对于逐点乘法优化，不同版本采用了不同的策略，值得注意的是，性能优化版本采用 Abdulrahman 等人[123,135] 所提出的非对称乘法（asymmetric multiplication）技术和对应的累加策略。对于内存优化，Huang 等人通过设计 *poly_half* 和 *polyvec_half* 数据结构来缓存中间计算结果，从而减少内存占用。

改进版 Plantard 模乘在 ARM Cortex-M4 处理器上的实现及其在 ML-KEM NTT 中的应用 算法 5.6 给出了 Huang 等人[119] 的改进版 Plantard 模乘在 ARM Cortex-M4 处理器上的实现，两个输入操作数中的 ζ 为提前已知的常量，该算法仅仅消耗 2 个 CPU 时钟周期。其中，`smulwb` 指

算法 5.5: ML-KEM NTT 中蒙哥马利模乘在 Cortex-M4 上的实现

输入: 两个 16 位有符号数 a, b 且满足 $ab \in [-q2^{l-1}, q2^{l-1})$

输出: $r_{top} = ab2^{-l} \bmod^{\pm} q$ 并且 $r_{top} \in (-q, q)$

1 **mul** c, a, b
2 **smulbb** $r, c, -q^{-1}$
3 **smlabb** r, r, q, c
4 **return** r_{top}

算法 5.6: ML-KEM NTT 中 Plantard 模乘在 Cortex-M4 上的实现

输入: $a \in [-137q, 230q]$; ζ 为常量; 预计算值 $\zeta q' = \zeta q^{-1} \bmod^{\pm} 2^{2l}$; $q = 3329$; $l = 16$; $\alpha = 3$

输出: $r_{top} = a\zeta(-2^{-2l}) \bmod^{\pm} q$ 并且 $r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **smulwb** $r, \zeta q', a$
2 **smlabb** $r, r, q, q2^{\alpha}$
3 **return** r_{top}

算法 5.7: ML-KEM NTT 中改进版 Plantard 约减在 Cortex-M4 上的实现

输入: 32 位有符号数 $c \in [q2^{l} - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$; $l = 16$; $\alpha = 3$

输出: $r_{top} = c(-2^{-2l}) \bmod^{\pm} q$ 并且 $r_{top} \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **mul** r, c, q'
2 **smlatb** $r, r, q, q2^{\alpha}$
3 **return** r_{top}

令用于计算 $r \leftarrow a\zeta q' \bmod^{\pm} 2^{2l} >> l$，计算结果保存在寄存器的低半部分。如果直接将 2^{α} 加到 r 的低半部分然后再与 q 相乘，那么可能会因为 r 的高半部分不为 0 而导致计算错误。因此使用 `smlabb` 来计算 $r_{\text{top}} \leftarrow (qr + q2^{\alpha}) >> l$，其中 $q2^{\alpha}$ 是提前已知的，可进行预计算，最终的计算结果保存在 r 的高半部分。

与之前已知最快的蒙哥马利模乘实现[121] 即算法 5.5 相比，算法 5.6 节省了一次乘法运算。Lyubashevsky 等人[136] 在利用 AVX2 指令实现蒙哥马利模乘时，也可以通过一些实现技巧来节省 1 条乘法指令，但高度依赖于 AVX2 指令集，无法应用到 Cortex-M4 处理器上。对于两个操作数均为未知值的模乘，先用 1 条乘法指令计算两者的乘积，即 $c = a \cdot b$，然后再使用 Plantard 约减即算法 5.7 对 c 进行模约减。

实现 ML-KEM 时，常用的策略是使用 CT 变换来实现 NTT，使用 GS 变换来实现 INTT。最近的研究表明[37,123]，NTT 和 INTT 都使用 CT 变换在某些场景下可以获得更佳的性能。算法 5.8 给出了 Huang 等人[119] 的基于改进版 Plantard 模乘算法所实现的双路 CT 蝴蝶变换，该算法用于计算 $a = (a_{top} + b_{top}\zeta) || (a_{bottom} + b_{bottom}\zeta), b = (a_{top} - b_{top}\zeta) || (a_{bottom} - b_{bottom}\zeta)$，其中 a, b 中均包含两个 16 位多项式系数，预计算表中的旋转因子的形式为 32 位整数 $\zeta \leftarrow (\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1} \bmod^{\pm} 2^{2l}$。该算法消耗 7 条指令，而使用蒙哥马利乘法则需要消耗 9 条指令。用于计算 INTT 的 GS 蝴蝶变换也可用类似的思路进行实现，同样消耗 7 条指令。

对于层合并策略，Huang 等人[119] 分析讨论了两种策略，分别为 3+3+1 和 4+3。得益于改

算法 5.8: ML-KEM NTT 中双路 CT 蝴蝶变换在 Cortex-M4 上的实现

输入: 两个 32 位有符号数 a, b，其中每个数中包含两个 16 位有符号数；32 位预计算格式的旋转因子 $\zeta \leftarrow (\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1} \bmod^{\pm} 2^{2l}$

输出: $a = (a_{top} + b_{top}\zeta)||(a_{bottom} + b_{bottom}\zeta), b = (a_{top} - b_{top}\zeta)||(a_{bottom} - b_{bottom}\zeta)$

1 **smulwb** t, ζ, b
2 **smulwt** b, ζ, b
3 **smlabb** $t, t, q, q2^{\alpha}$
4 **smlabb** $b, b, q, q2^{\alpha}$
5 **pkhtb** t, b, t, asr#16
6 **usub16** b, a, t
7 **uadd16** a, a, t
8 **return** a, b

算法 5.9: ML-KEM NTT 中 Plantard 模乘在 RV32IM 上的实现

输入: $a \in [-137q, 230q]$；ζ 为常量；预计算值 $\zeta q' = \zeta q^{-1} \bmod^{\pm} 2^{2l}$；$q = 3329$；$l = 16$；$\alpha = 3$

输出: $r = a\zeta(-2^{-2l}) \bmod^{\pm} q$ 并且 $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **mul** $r, a, \zeta q'$；/* $r \leftarrow a\zeta q' \bmod^{\pm} 2^{2l}$ */
2 **srai** r, r, #16
3 **addi** $r, r, 2^{\alpha}$
4 **mulh** $r, r, q2^{l}$
5 **return** r

算法 5.10: ML-KEM NTT 中 CT 蝴蝶变换在 RV32IM 上的实现

输入: 两个有符号数 a, b；32 位旋转因子 ζ

输出: $a' = a + b\zeta, b' = a - b\zeta$

1 **mul** b, b, ζ
2 **srai** b, b, #16
3 **addi** $b, b, 2^{\alpha}$
4 **mulh** $t, b, q2^{l}$
5 **sub** b', a, t
6 **add** a', a, t
7 **return** a', b'

进版 Plantard 模乘算法的输入输出范围的优势，Huang 等人设计了比之前工作更优的延迟约减策略。除此以外，Huang 等人还实现了双路 Plantard 约减，其只消耗 5 个时钟周期，而基于蒙哥马利约减和 Barrett 约减的实现分别消耗 7 个和 8 个时钟周期。

改进版 Plantard 模乘在单发射 RV32IM 处理器上的实现及其在 ML-KEM NTT 中的应用 算法 5.9 给出了 Huang 等人[132] 的改进版 Plantard 模乘在单发射 RV32IM 处理器上的实现，使用 32×32 位乘法指令 `mul` 来计算 $a\zeta q' \bmod^{\pm} 2^{2l}$。RISC-V 不支持形如 ARM Cortex-M3 上的 barrel 移位操作，因此需要 1 条移位指令来计算 $a\zeta q' \bmod^{\pm} 2^{2l} >> l$。直接计算 $rq >> l$ 需使用 1 条乘法指令和 1 条移位指令，Huang 等人的优化思路是预计算 $q2^{l} \leftarrow q \times 2^{l}$，考虑到 $rq >> l$ 与 $rq2^{l} >> 2l$ 等价，因此可通过 1 条 `mulh` 指令来实现 $rq2^{l} >> 2l$，而无需使用额外的移位指令。最终，算法 5.9 消耗 4 条指令，其中包括 2 条乘法指令、1 条移位指令和 1 条加法指令，而目前

已知的最快蒙哥马利模乘实现需要使用 5 条指令[36]。

实现 NTT 时，旋转因子的预计算格式为 $((\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1}) \bmod^{\pm} 2^{2l}$。算法 5.10 给出了 CT 蝴蝶变换在单发射 RV32IM 处理器上的实现，GS 蝴蝶变换的实现与之类似，CT 与 GS 蝴蝶变换均消耗 6 条指令。考虑到 CT 与 GS 蝴蝶变换的开销相同，因此使用 CT 蝴蝶变换实现 NTT，使用 GS 蝴蝶变换实现 INTT。

对于层合并策略，Huang 等人[132] 分别使用 4+3 和 3+4 层合并策略来实现 NTT 和 INTT。与 ARM Cortex-M3 上的实现类似，Huang 等人也分别实现了内存优化和性能优化版本，并依次设计了更优的延迟约减策略。对于逐点乘法和内存优化技术，与 ARM Cortex-M3 上的实现类似。

改进版 Plantard 模乘在 ML-DSA 方案中的应用 对于 ML-DSA 签名（见算法 2.22）中的多项式乘法 $c\mathbf{s_0}$ 和 $c\mathbf{s_1}$，其结果多项式的系数均小于 $\beta = \tau \cdot \eta$。根据 Chung 等人的研究[137] 可知，多项式乘法 $c\mathbf{s}_i$ 可使用多项式环 $\mathbb{Z}_{q'}[X]/(X^n+1)$ 上的 NTT 算法来实现，其中 $q' > 2\beta$。鉴于 q' 一般小于 2^{16}，因此 $c\mathbf{s}_i$ 可使用 16 位 NTT 实现。由表 2.2 可知，β 对于 3 个 ML-DSA 参数集的取值分别为 78、196 和 120。Abdulrahman 等人[138] 在 ARM Cortex-M4 处理器上使用类似的思路对 $c\mathbf{s}_i$ 的计算进行了加速，对于 ML-DSA-65，他们使用 $\mathbb{Z}_{769}[X]/(X^n+1)$ 上的 NTT 算法；对于 ML-DSA-{44,87}，他们使用 $\mathbb{Z}_{257}[X]/(X^n-2^{2^3})$ 上的 FNT（Fermat Number Transform）算法。

ML-DSA 签名即算法 2.22 的 `while` 循环中，逐点乘法与 INTT 的使用次数多于 NTT；而且 Abdulrahman 等人所实现的 $\mathbb{Z}_{257}[X]/(X^n-2^{2^3})$ 上的 FNT 算法能更快地计算逐点乘法。因此，对于 ARM Cortex-M4 上的 $c\mathbf{s}_i$ 计算，Huang 等人[133] 直接复用 Abdulrahman 等人的策略，并且使用改进版 Plantard 模乘来加速 $\mathbb{Z}_{769}[X]/(X^n+1)$ 上的 NTT 算法。

对于 ARM Cortex-M3 上的实现，Huang 等人[133] 使用 $\mathbb{Z}_{769}[X]/(X^n+1)$ 上的 NTT 来计算 ML-DSA 的 $c\mathbf{s}_i$。正如本章 5.2.1 所述，ARM Cortex-M3 处理器的 32 位乘法指令是非恒定执行时间的，ML-DSA 密钥生成（算法 2.20）和签名（算法 2.22）涉及到私钥相关数据的计算，因此需要保证执行时间是恒定的。因此除了上述用于计算 $c\mathbf{s}_i$ 的 16 位 NTT 以及下文即将介绍的 $c\mathbf{t}_i$，其余的 NTT 实现中所需的 32 位乘法均使用 16 位乘法指令进行构造，从而保证其执行时间是恒定的。

对于 ML-DSA 签名（算法 2.22）中的 $c\mathbf{t}_i$，结果多项式的系数均小于 $\tau 2^{12} \le 60 \times 2^{12} = 245760$。因此 $c\mathbf{t}_i$ 可使用 $\mathbb{Z}_{q'}/[X](X^n+1)$ 上的 32 位 NTT 算法来实现，其中 $q' > 2\beta' = 491520$。Huang 等人[133] 使用多模数 NTT 来计算 $c\mathbf{t}_i$，并且 $q' = 769 \times 3329 = 2560001$，其中 $q' > 2\beta'$ 显然成立。多模数 NTT 是指，对于 $q' = q_0 \times q_1$、$q_0 = 769$、$q_1 = 3329$，分别计算 $\mathbb{Z}_{q_0}[X]/(X^n+1)$ 和 $\mathbb{Z}_{q_1}[X]/(X^n+1)$ 上的 16 位 NTT，然后使用中国剩余定义（CRT）求得 $\mathbb{Z}_{q'}[X]/(X^n+1)$ 上的多项式乘法结果。类似的优化思路曾被用于加速 Saber、NTRU 和 LAC 方案[137,139]。使用多模数 NTT 来加速 $c\mathbf{t}_i$ 曾在文献[34] 中给出过相关讨论，其中提到他们基于蒙哥马利模乘实现的多模数

NTT 无法优于 32 位 NTT 实现。Huang 等人[133] 则使用改进版 Plantard 模乘来实现多模数 NTT，最终的实验表明其比 32 位 NTT 实现快。

模数 $q_0 = 769$ 和 $q_1 = 3329$ 满足 $n|(q_i - 1)$，但不满足 $2n|(q_i - 1)$，其中 $i = 0, 1$、$n = 256$。$\mathbb{Z}_{q_i}$ 上仅存在 n 次本原根 ζ_i，不存在 2n 次本原根；因此，分圆多项式 $X^n + 1$ 可被分解为阶为 1 因子的乘积；这也意味着 NTT 与 INTT 最多可计算 7 层。由中国剩余定理可知，$\mathbb{Z}_{q_i}[X]/(X^n + 1) \cong \prod \mathbb{Z}_{q_i}[X]/(X^2 - \zeta_i^j)$，其中 $j = 1, 3, 5, \cdots, 255$。相比于 8 层 NTT 实现，7 层实现中 NTT 与 INTT 的计算更快，而且可使用延迟约减和非对称乘法策略[123,135] 来加速逐点乘法。

在 ML-DSA 签名（即算法 2.22）中，$c\mathbf{t}_0$ 的计算使用多模数 NTT 进行加速，其中涉及到 $\mathbb{Z}_{769}[X]/(X^n + 1)$ 和 $\mathbb{Z}_{3329}[X]/(X^n + 1)$ 上的 NTT，$\mathbb{Z}_{769}[X]/(X^n + 1)$ 上的 NTT 也被用于签名过程中的 $c\mathbf{s}_i$ 计算。

在 ML-DSA 验证（算法 2.21）中，$\mathbf{Az} - c\mathbf{t}_1 \times 2^d$ 的计算使用 32 位 NTT 进行实现。在 32 位 NTT 实现中，计算完逐点乘法即可计算多项式减法，因为所有值均处于同一 NTT 域中。然而，如果使用多模数 NTT 来计算 $c\mathbf{t}_1$，则不能计算完逐点乘法后直接计算多项式减法，因为他们在不同的 NTT 域内，因此需要对一个多项式向量执行额外的 INTT 操作。除此以外，在 ML-DSA 验证过程中，c 和 $\mathbf{t}_1$ 均为公开值，可使用 Cortex-M3 上的非恒定时间的 32 位 NTT 实现进行计算。即使多模数 NTT 比非恒定时间的 32 位 NTT 快，但考虑到额外的 INTT 引入的开销，多模数 NTT 将不具有性能优势。总而言之，Huang 等人[133] 的多模数 NTT 仅用于加速 ML-DSA 签名过程中的 $c\mathbf{t}_0$ 计算。

5.3 改进版 Plantard 模乘算法在 ML-KEM 方案中的实现和应用

本节给出针对 ML-KEM 方案的改进版 Plantard 模乘算法以及对应的 NTT 多项式乘法在双发射 RV32IM 和 RV64IM 处理器上的优化方法。对于 ML-KEM 方案，模数 $q = 3329$，因此 l 设为 16；为了使 $q < 2^{l-\alpha-1}$ 成立，α 取值为 3。实现 NTT 多项式乘法时，模乘的两个操作数之一是提前已知的旋转因子（twiddle factor），并且其范围是 $[0, q)$。考虑到改进版 Plantard 模乘算法即算法 5.2 所允许的 ab 的范围是 $[q2^l - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$，那么所允许的 a 的最大值与最小值分别为：

$$a_{max} < (2^{2l} - q2^{l+\alpha})/b_{max} = (2^{32} - 3329 \times 2^{19})/3328 \approx 230.13q$$
$$a_{min} > (q2^l - q2^{l+\alpha})/b_{max} = (3329 \times 2^{16} - 3329 \times 2^{19})/3328 \approx -137.85q$$

因此，所允许的 a 的输入范围为 $[-137q, 230q]$。

```
.macro ct_bfu_x4 a0_0,a0_1,a1_0,a1_1,a2_0,a2_1,a3_0,a3_1,zeta0,zeta1,zeta2,
    zeta3,q16,t0,t1,t2,t3
    mul  \t0, \a0_1, \zeta0;     mul  \t1, \a1_1, \zeta1
    mul  \t2, \a2_1, \zeta2;     mul  \t3, \a3_1, \zeta3
    srai \t0, \t0, 16;           srai \t1, \t1, 16
    srai \t2, \t2, 16;           srai \t3, \t3, 16
    addi \t0, \t0, 8;            addi \t1, \t1, 8
    addi \t2, \t2, 8;            addi \t3, \t3, 8
    mulh \t0, \t0, \q16;         mulh \t1, \t1, \q16
    mulh \t2, \t2, \q16;         mulh \t3, \t3, \q16
    sub  \a0_1, \a0_0, \t0;      sub  \a1_1, \a1_0, \t1
    sub  \a2_1, \a2_0, \t2;      sub  \a3_1, \a3_0, \t3
    add  \a0_0, \a0_0, \t0;      add  \a1_0, \a1_0, \t1
    add  \a2_0, \a2_0, \t2;      add  \a3_0, \a3_0, \t3
.endm
```

代码片段 5.1　RV32IM 上基于改进版 Plantard 模乘的 4 路交替执行的 CT 蝴蝶变换

5.3.1　双发射 RV32IM 处理器上的实现

双发射 RV32IM 处理器上的 NTT 多项式乘法实现与本章 5.2.3 所述的单发射 RV32IM 上的实现基本一致，即算法 5.9 和算法 5.10，区别在于本节将重点关注双发射流水线优化，因此下文将重点讲述双发射流水线优化方法。

双发射流水线优化方法　下文以算法 5.9 为例阐述双发射优化方法，该算法中每一条指令都依赖于上一条指令的计算结果，比如第二行的 `srai` 指令依赖于第一行 `mul` 指令的计算结果 r。这种实现方法对于单发射处理器是没问题的，但在双发射处理器上却会产生 RAW（Read After Write）数据冒险。具体地，理想情况下第一行的 `mul` 指令应和第二行的 `srai` 指令并行执行，但由于数据依赖的存在，导致 `srai` 不得不等待 `mul` 的计算结果，因此导致了流水线停顿。优化方法是将多个 Plantard 模乘或 CT 蝴蝶变换交替执行，从而缓解上述 RAW 数据冒险。代码片段 5.1 展示了基于本章改进版 Plantard 模乘的 4 路交替执行的 CT 蝴蝶变换。由表 5.1 可知，乘法指令的时延为 3 或 4 个时钟周期，因此 6 路或 8 路交替执行能完全消除 RAW 数据冒险。然而，在 NTT 实现中，由于寄存器数量的限制，最多只能实现 4 路交替执行。每个基于改进版 Plantard 的 CT 蝴蝶变换需要 2 条乘法指令和 4 条逻辑/算术指令，其理想 CPI 为 $(1\times2+0.5\times4)/6 = 0.67$。我们的测试表明，4 路交替执行的 CT 蝴蝶变换在 RV32IM 上的 CPI 为 0.88；在 RV64IM 上的 CPI 为 1.08。RVV 实现的双发射优化是类似的，考虑到可使用标量寄存器来加载常量，并可利

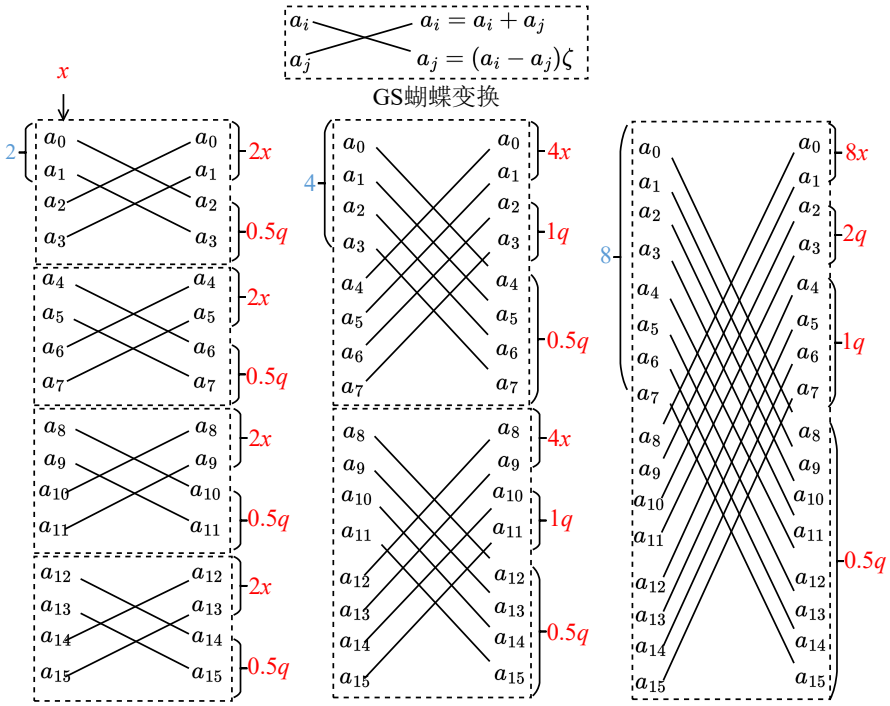


图 5.1　32 位 RISC-V 上 GS 蝴蝶变换实现 INTT 时前 16 个多项式系数的范围变化，a_i 和 a_j 表示多项式系数，x 和 $2x$ 表示系数绝对值的最大值[132]

用 `vmul{h}.vx` 指令来计算向量乘法，因此可最多实现 8 路交替执行，最终使 RVV 实现的 CPI 接近理想值 1.0。

NTT/INTT 中的蝴蝶变换实现　实现思路与算法 5.10 类似，区别在于双发射 RV32IM 处理器上进行了 4 路交替执行来消除 RAW 数据冒险，交替执行的基本方法见代码片段 5.1。本节使用 CT 蝴蝶变换来实现 NTT，使用 GS 蝴蝶变换来实现 INTT。对于使用 CT 蝴蝶变换来实现 INTT，其性能弱于 GS 蝴蝶变换，具体原因包括：（1）使用 CT 蝴蝶变换实现 INTT 会使旋转因子的预计算表大小加倍，这会增加内存访问开销；（2）结合下文的延迟约减策略，用 GS 蝴蝶变换来实现 INTT 也足够高效。

层合并策略　RISC-V 指令集共有 30 个寄存器可供编程使用，因此一次最多可加载 16 个多项式系数来实现 4 层 NTT/INTT 合并策略，其余 14 个寄存器用于循环控制、保存旋转因子或作为临时寄存器使用。因此，本工作分别采用 4+3 和 3+4 层合并策略来实现 NTT 和 INTT。

延迟约减　与 ARM Cortex-M3 处理器上的实现类似，RISC-V 上也可使用 32 位有符号数表示多项式系数，并考虑到算法 5.9 所允许的最大输入范围是 $[-137q, 230q]$，因此相比于 ARM Cortex-M4 上的实现可设计更优的延迟约减策略。使用 CT 蝴蝶变换实现 NTT 时的延迟约减策略与

算法 5.11: ML-KEM NTT 中改进版 Plantard 模乘在 RV64IM 上的实现

输入: $a \in [-137q, 230q]$；ζ 为常量；预计算值 $\zeta q' = \zeta q^{-1} \bmod^{\pm} 2^{2l}$；$q = 3329$；$l = 16$；$\alpha = 3$

输出: $r = a\zeta(-2^{-2l}) \bmod^{\pm} q$ 并且 $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **mulw** $r, a, \zeta q'$; /* $r \leftarrow a\zeta q' \bmod^{\pm} 2^{2l}$ */
2 **srai** r, r, #16
3 **addi** $r, r, 2^{\alpha}$
4 **mulh** $r, r, q2^{48}$
5 **return** r

算法 5.12: ML-KEM NTT 中 CT 蝴蝶变换在 RV64IM 上的实现

输入: 两个有符号数 a, b；32 位旋转因子 ζ

输出: $a' = a + b\zeta, b' = a - b\zeta$

1 **mulw** b, b, ζ
2 **srai** b, b, #16
3 **addi** $b, b, 2^{\alpha}$
4 **mulh** $t, b, q2^{48}$
5 **sub** b', a, t
6 **add** a', a, t
7 **return** a', b'

ARM Cortex-M4 处理器上相同，因此不再赘述。下文将重点描述用 GS 蝴蝶变换实现 INTT 时的延迟约减策略。因为采用 3+4 层合并策略，前 3 层 INTT 计算完成后需将多项式系数保存至 16 位有符号数中，因此需要分析这些多项式系数范围是否超过了 16 位有符号数的表示范围。

图 5.1 给出了 GS 蝴蝶变换实现 INTT 时前 16 个系数的范围变化示意图，INTT 的输入多项式系数范围为 $(-q/2, q/2)$，经过 3 层计算后，前 2 个系数的最大值变为了 $4q$，不会超出 16 位有符号数的表示范围。再经过 4 层计算后，系数最大值达到 $64q$，仍然处于改进版 Plantard 模乘所允许的输入范围内。因此 GS 蝴蝶变换实现的 INTT 中无需执行任何额外的模约减操作。

逐点乘法优化 经过 7 层 NTT 运算后，多项式 a 和 b 被变换到 NTT 域上，即 $\hat{a}$ 和 $\hat{b}$，然后执行 $\mathbb{Z}_q[X]/(X^2 - \zeta^{2\mathrm{br}_7(i)+1})$ 上的逐点乘法（pointwise multiplication）$\hat{c} = \hat{a} \circ \hat{b}$，其中 $\mathrm{br}_7(i)$ 表示 7 位无符号整数的比特逆操作，$\circ$ 符号表示逐点乘法，由 128 个运算组成，每个运算为：$\hat{c}_{2i} + \hat{c}_{2i+1}X = (\hat{a}_{2i} + \hat{a}_{2i+1}X)(\hat{b}_{2i} + \hat{b}_{2i+1}X) \bmod (X^2 - \zeta^{2\mathrm{br}_7(i)+1})$，其中 $\hat{c}_{2i} = \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$，$\hat{c}_{2i+1} = \hat{a}_{2i}\hat{b}_{2i+1} + \hat{b}_{2i}\hat{a}_{2i+1}$。

本工作采用 Abdulrahman 等人[123,135] 所提出的非对称乘法（asymmetric multiplication）技术和对应的累加策略，其可以减少模乘与模约减操作的次数，但会增加内存占用。在计算 ML-KEM 中的矩阵向量乘法 $\mathbf{As}$ 时，NTT 域的向量 $\hat{\mathbf{s}}$ 被复用 k 次，在计算逐点乘法 $\hat{a} \circ \hat{s} = \hat{c}$ 时，$\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ 也需要被计算 k 次。非对称乘法技术是指将 $\hat{s}_{2i+1}\zeta^{2\mathrm{br}_7(i)+1}$ 缓存至 $\hat{\mathbf{s}}'$ 中。当矩阵 $\hat{\mathbf{A}}$ 的第一行与 $\hat{\mathbf{s}}$ 相乘时，计算 $\hat{s}_{2i+1} \cdot \zeta^{2\mathrm{br}_7(i)+1} \bmod q$ 并缓存至 $\hat{\mathbf{s}}'$ 中，其中的模乘运算可使用改进版 Plantard 模乘进行实现。后续矩阵 $\hat{\mathbf{A}}$ 的其余 $k-1$ 行与 $\hat{\mathbf{s}}$ 相乘时，直接使用 $\hat{\mathbf{s}}'$，而无需重复计算 $\hat{s}_{2i+1} \cdot \zeta^{2\mathrm{br}_7(i)+1} \bmod q$。

算法 5.13: ML-KEM NTT 中蒙哥马利模乘在 RVV 上的实现

输入: 向量寄存器 va；16 位旋转因子 $\zeta' \leftarrow \zeta \cdot 2^{16} \bmod q$；预计算值 $\zeta' q^{-1} \leftarrow \zeta' \cdot q^{-1} \bmod 2^{16}$

输出: $vr = va \cdot \zeta \bmod^{\pm} q$

1 **vmul.vx** $vt_0, va, \zeta' q^{-1}$
2 **vmulh.vx** vt_1, va, ζ'
3 **vmulh.vx** vt_0, vt_0, q
4 **vsub.vv** vr, vt_1, vt_0
5 **return** vr

算法 5.14: ML-KEM NTT 中 CT 蝴蝶变换在 RVV 上的实现

输入: 两个向量寄存器 va, vb；16 位旋转因子 $\zeta' \leftarrow \zeta \cdot 2^{16} \bmod q$；预计算值 $\zeta' q^{-1} \leftarrow \zeta' \cdot q^{-1} \bmod 2^{16}$

输出: $va' = va + vb \cdot \zeta, vb' = va - vb \cdot \zeta$

1 **vmul.vx** $vt_0, vb, \zeta' q^{-1}$
2 **vmulh.vx** vt_1, vb, ζ'
3 **vmulh.vx** vt_0, vt_0, q; **vsub.vv** vt_0, vt_1, vt_0
4 **vsub.vv** vb', va, vt_0; **vadd.vv** va', va, vt_0
5 **return** va', vb'

5.3.2 双发射 RV64IM 处理器上的实现

NTT/INTT 中的蝴蝶变换实现 RV64IM 上的改进版 Plantard 以及对应的 CT 蝴蝶变换实现见算法 5.11 和算法 5.12，相比于 RV32IM 上的实现，区别在于：（1）第一行的乘法指令使用 32 位乘法指令 `mulw` 而不是 64 位乘法指令 `mul`，因为 RV64IM 上前者仅需 3 个时钟周期，而后者需要 4 个；（2）第四行所使用的常数为 $q2^{48}$ 而不是 RV32IM 上的 $q2^{16}$。双发射流水线优化思路与 RV32IM 上相同，通过尽可能地交替执行来缓解 RAW 数据冒险，交替执行的基本方法见代码片段 5.1。层合并策略、延迟约减和逐点乘法优化方法与本章 5.3.1 保持一致，此处不再赘述。

5.3.3 RVV 处理器上的实现

由于改进版 Plantard 模乘需要使用 16×32 比特乘法器，而 RVV（RISC-V Vector）所支持的向量乘法指令均为 $l \times l$ 比特格式，其中 $l \in \{8, 16, 32, 64\}$，如果使用 32×32 比特乘法器来实现 16×32 比特乘法，那么相比于 16×16 比特方法并行性减半。因此，在 RVV 上我们使用蒙哥马利模乘方法来实现 NTT 多项式乘法。

蒙哥马利模乘实现与对应的 CT 蝴蝶变换实现见算法 5.13 和算法 5.14，向量引擎的 SEW（Selected Element Width）被配置为 16 比特、LMUL 被配置为 1。其中的实现技巧是尽可能地使用 `vmul{h}.vx` 指令，即尽可能地使用标量寄存器来保存 ζ' 和 $\zeta' q^{-1}$，这一技巧能降低向量寄存器的分配压力。

所使用的层合并策略是 1+6 层合并，NTT 计算过程中无需额外的模约减。对于 INTT，计算第 1 层前每个系数都被模乘了常量 $\mathsf{mont}^2/128$ 从而将系数范围均统一为 $(-q, q)$，其中 mont^2 用于蒙哥马利域的转换。完成 3 层 INTT 计算后，16 个向量寄存器中的值接近溢出边界，因此需要进行模约减；再计算 2 层后，2 个向量寄存器需要模约减；因此，INTT 中共需要对 18 个向量寄存器执行模约减。

算法 5.15: ML-DSA NTT 中改进版 Plantard 模乘在 RV64IM 上的实现

输入: $a \in [-130687q, 131456q]$；$\zeta$ 为常量；$\zeta q' \leftarrow \zeta q^{-1} \bmod^{\pm} 2^{2l}$；$q = 8380417$；$l = 32$；$\alpha = 8$

输出: $r = a\zeta(-2^{-2l}) \bmod^{\pm} q$ 并且 $r \in [-\frac{q+1}{2}, \frac{q}{2})$

1 **mul** $r, a, \zeta q'$；/* $r \leftarrow a\zeta q' \bmod^{\pm} 2^{2l}$ */
2 **srai** r, r, #32
3 **addi** $r, r, 2^{\alpha}$
4 **mulh** $r, r, q2^{l}$
5 **return** r

算法 5.16: ML-DSA NTT 中 CT 蝴蝶变换在 RV64IM 上的实现

输入: 两个有符号数 a, b；64 位旋转因子 ζ

输出: $a' = a + b\zeta, b' = a - b\zeta$

1 **mul** b, b, ζ
2 **srai** b, b, #32
3 **addi** $b, b, 2^{\alpha}$
4 **mulh** $t, b, q2^{l}$
5 **sub** b', a, t
6 **add** a', a, t
7 **return** a', b'

除此以外，NTT/INTT 实现过程中，需使用 `vrgather.vv` 和 `vmerge.vvm` 指令来重排系数顺序，从而构造所需的执行流程。逐点乘法的实现与 AVX2 上的实现[1]类似，因此不再赘述。

5.4 改进版 Plantard 模乘算法在 ML-DSA 方案中的实现和应用

考虑到 ML-DSA 方案的模数为 $q = 8380417$ 且 $2^{22} < q < 2^{23}$，因此其 NTT 计算需使用 32 位乘法指令，称这样的 NTT 为 32 位 NTT。使用改进版 Plantard 模乘算法实现 ML-DSA 的 32 位 NTT 需要使用 32×64 位乘法指令，而 ARM Cortex-M3、Cortex-M4 和 RV32IM 均不存在这样的乘法指令。RV64IM 支持 64×64 位乘法指令，因此可用于实现上述 32 位 NTT。下文介绍本工作在双发射 RV64IM 处理器上 ML-DSA NTT 的优化实现、在双发射 RV32IM 处理器和 RVV 上实现 ML-DSA NTT 的方法。

5.4.1 双发射 RV64IM 处理器上的实现

本节给出针对 ML-DSA 方案的改进版 Plantard 模乘算法以及对应的 NTT 多项式乘法在双发射 RV64IM 处理器上的实现。对于 ML-DSA 方案，模数 $q = 8380417$，因此 l 设为 32；为了使 $q < 2^{l-\alpha-1}$ 成立，α 取值为 8。

实现 NTT 多项式乘法时，模乘的两个操作数之一是提前已知的旋转因子（twiddle factor），并且其范围是 $[0, q)$。考虑到算法 5.2 所允许的 ab 的范围是 $[q2^{l} - q2^{l+\alpha}, 2^{2l} - q2^{l+\alpha})$，那么所允许的 a 的最大值与最小值为：

$$a_{max} < (2^{2l} - q2^{l+\alpha})/b_{max} = (2^{64} - 8380417 \times 2^{40})/8380417 \approx 131456.58q$$

$$a_{min} > (q2^{l} - q2^{l+\alpha})/b_{max} = (8380417 \times 2^{32} - 8380417 \times 2^{40})/8380417 \approx -130687.61q$$

[1]https://github.com/pq-crystals/kyber/blob/8e390f7152cf66f27cb39f164a3b2a8256bf863c/avx2/ntt.S

因此，所允许的 a 的输入范围为 $[-130687q, 131456q]$。

改进版 Plantard 模乘实现 算法 5.15 给出了本工作改进版 Plantard 模乘在 RV64IM 处理器上的实现，该算法使用 64×64 位乘法指令 `mul` 来计算 $a\zeta q' \bmod^{\pm} 2^{2l}$。该算法与 RV32IM 处理器上的实现（算法 5.9）相似，区别在于此处 $l = 32$、$\alpha = 8$。该算法消耗 4 条指令，其中包括 2 条乘法指令、1 条移位指令和 1 条加法指令。

NTT/INTT 中的蝴蝶变换实现 算法 5.16 给出了基于改进版 Plantard 模乘的 CT 蝴蝶变换在 RV64IM 处理器上的实现，其中旋转因子的预计算格式为 $((\zeta \cdot (-2^{2l}) \bmod q) \cdot q^{-1}) \bmod^{\pm} 2^{2l}$。GS 蝴蝶变换的实现与之类似，CT 与 GS 蝴蝶变换均消耗 6 条指令。与本章 5.2.3 中 ML-KEM NTT 在 RV32IM 上的实现相同，分别使用 CT 和 GS 蝴蝶变换实现 NTT 和 INTT。

双发射流水线优化 此处的双发射流水线优化与本章 5.3.1 中的优化方法相同，即通过交替执行多个并行的蝴蝶变换来缓解流水线停顿，具体的交替执行方法见代码片段 5.1。

层合并策略 与本章 5.2.3 中的实现思路类似，得益于 RISC-V 指令集提供的 30 个可用寄存器，最多可实现 4 层 NTT/INTT 合并，因此采用 4+4 层合并策略来实现 NTT 和 INTT。

延迟约减 RV64IM 上的实现允许中间值临时地超出 32 位有符号数的范围，但仍需在 64 位有符号数范围内，并需保证多项式系数写回内存前将其约减到 32 位有符号数范围内。对于使用 CT 蝴蝶变换实现的 NTT，输入范围为 $(-q, q)$，经过 8 层计算后，范围变为 $(-9q, 9q)$，仍处于 32 位有符号数范围内，因此 NTT 计算过程无需执行模约减操作。逐点乘法的输入来自于 NTT 的输出，范围为 $(-9q, 9q)$，处于算法 5.15 所允许的范围内；逐点乘法的输出范围为 $[-\frac{q+1}{2}, \frac{q}{2})$。对于使用 GS 蝴蝶变换实现的 INTT，其输入来自于逐点乘法的输出，范围为 $[-\frac{q+1}{2}, \frac{q}{2})$，经过 8 层计算后，系数最大值为 $2^7 q$，仍处于 32 位有符号数范围内，因此 INTT 计算过程也无需执行模约减操作。

5.4.2 双发射 RV32IM 和 RVV 处理器上的实现

ML-DSA NTT 在 RV32IM 和 RVV 上无法使用改进版 Plantard 模乘，因为其需要使用 32×64 比特乘法指令。因此本节使用蒙哥马利模乘来实现 RV32IM 和 RVV 上的 ML-DSA NTT 多项式乘法。

算法 5.17 和算法 5.18 分别给出了 RV32IM 和 RVV 上的蒙哥马利模乘实现，考虑到 NTT 实现中模乘的操作数之一为旋转因子常量，因此预计算了 $bq' \leftarrow bq^{-1} \bmod^{\pm} \beta$，从而节省了 1 条乘

算法 5.17: ML-DSA NTT 中的蒙哥马利模乘在 RV32IM 上的实现

输入: a 和 b 且满足 $-\frac{\beta}{2}q \le ab < \frac{\beta}{2}q$；$q \in (0, \frac{\beta}{2})$；$\beta = 2^{32}$；$b$ 为常量且 $b \in (0, q)$；预计算值 $bq' \leftarrow bq^{-1} \bmod^{\pm} \beta$

输出: $r = ab\beta^{-1} \bmod^{\pm} q$ 且 $r \in (-q, q)$

1 **mul** r, a, bq'
2 **mulh** t, a, b
3 **mulh** r, r, q
4 **sub** r, t, r
5 **return** r

算法 5.18: ML-DSA NTT 中的蒙哥马利模乘在 RVV 上的实现

输入: 向量寄存器 va，其中包含 4 个 32 位有符号数；32 位旋转因子 $\zeta' \leftarrow \zeta \cdot 2^{32} \bmod q$；预计算值 $\zeta' q^{-1} \leftarrow \zeta' \cdot q^{-1} \bmod 2^{32}$

输出: $vr = va \cdot \zeta \bmod^{\pm} q$

1 **vmul.vx** $vt_0, va, \zeta' q^{-1}$
2 **vmulh.vx** vt_1, va, ζ'
3 **vmulh.vx** vt_0, vt_0, q
4 **vsub.vv** vr, vt_1, vt_0
5 **return** vr

法指令；这一技巧也在 Kyber 的 AVX2 实现[1]中被用到。以算法 5.17 为例，其共计消耗 3 条乘法指令 `mul{h}` 和 1 条减法指令。

然而，预计算 bq' 虽然能减少 1 条乘法指令，但也增加了寄存器分配压力，从而导致我们无法在 RV32IM 上实现 4 层合并，而最多只能实现 3 层合并，因此最终我们的 NTT/INTT 实现的层合并策略为 3+3+2。在 RVV 上，我们不仅可以使用 32 个向量寄存器，还能使用标量寄存器来加载部分常量，从而使用形如 `vmul{h}.vx` 的指令计算乘法，其中一个操作数为向量寄存器，另一个操作数为标量寄存器。这降低了向量寄存器的分配压力，使得我们最多能实现 4 层合并，因此最终 NTT/INTT 实现的层合并策略为 4+4。与上文的分析类似，RV32IM 和 RVV 上的 NTT/INTT 的计算过程均无需执行模约减操作。

5.5 Saber 方案的多项式乘法优化

正如本文 2.6 所述，Saber 方案的多项式环无法直接使用 NTT 多项式乘法，其参考实现中使用了 Toom-Cook 和 Karatsuba 结合的算法来计算多项式乘法。2021 年，Chung 等人[37] 为 Saber 方案适配了 NTT 多项式乘法，并且相比于其他实现方法具有明显的性能优势。

5.5.1 NTT 多项式乘法的参数设计

为了给 Saber 方案适配 NTT 多项式乘法，我们需要选择一个素数 M，其必须大于 Saber 多项式乘法在不进行模约减的情况下所能产生的最大值。Saber 的三个不同安全级别中参数 μ 的取值分别为 10、8 和 6，这意味着诸如算法 2.17 中多项式向量 $\mathbf{s}$ 中的系数范围分别为 $[-5, 5]$、$[-4, 4]$ 和 $[-3, 3]$。为了得到更紧凑的范围表示，我们将 $\mathbb{Z}_q$ 中的多项式系数转换为有符号表示，其范围表示为 $[-\frac{q}{2}, \frac{q}{2}]$。令 a 为 R_q 中的多项式，s 为 Saber 中的噪声多项式，考虑多项式乘法

[1] https://github.com/pq-crystals/kyber/blob/8e390f7152cf66f27cb39f164a3b2a8256bf863c/avx2/ntt.S

$a \cdot s$，在不进行模约减的情况下，其产生的最大值与最小值分别为 $5 \cdot \frac{q}{2} \cdot n$ 和 $-5 \cdot \frac{q}{2} \cdot n$，因此我们得到 M 的第一个约束条件：

$$M \geq 2 \cdot 5 \cdot \frac{q}{2} \cdot n = 10485760 \tag{5.2}$$

在使用 NTT 实现 Saber 的多项式乘法时，前向 NTT 使用 CT 蝴蝶变换，逆向 NTT（INTT）使用 GS 蝴蝶变换。对于两个整数 a 和 b，CT 蝴蝶变换计算 $a+\gamma \cdot b$ 和 $a-\gamma \cdot b$，GS 蝴蝶变换计算 $a+b$ 和 $(a-b) \cdot \gamma$。考虑使用蒙哥马利模乘计算 $\gamma \cdot b$ 和 $(a-b) \cdot \gamma$，蒙哥马利模乘的计算结果的范围为 $(-M, M)$。因此，经过一次 CT 蝴蝶变换后，系数范围增加 M，经过一次 GS 蝴蝶变换后，系数范围变为原来的 2 倍。

假设输入范围为 $(-x, x)$，经过 y 层 GS 蝴蝶变换后，输出范围为 $(-x \cdot 2^y, x \cdot 2^y)$。考虑使用 32 位有符号表示法，如果 $x \cdot 2^y > 2^{31}$，那么需要执行模约减来避免数据溢出。为了尽可能消除 GS 蝴蝶变换实现中的模约减，因此得到 M 的第二个约束条件：

$$x \cdot 2^y < 2^{31} \tag{5.3}$$

其中 x 为 INTT 输入的最大值，y 为 INTT 的计算层数。

本工作的实现采用 6 层 NTT 多项式乘法，INTT 的输入来源于逐点乘法，数据范围为 $(-2M, 2M)$，因此最终 M 的范围约束为：

$$M \geq 2 \cdot 5 \cdot \frac{q}{2} \cdot n \text{ 并且 } 2M \cdot 2^6 < 2^{31} \tag{5.4}$$

因此得到 $5 \cdot q \cdot n \leq M < 2^{24}$，最终的选择为 $M = 10487809 = 512 \cdot 20484 + 1$，其满足 $2n \mid (M-1)$，因此 $\mathbb{Z}_M$ 中存在 $2n$ 次本原根，既可以实现 8 层完整 NTT，也可以实现各种不完整 NTT。

5.5.2 不完整 NTT 性能分析

Lyubashevsky 等人的研究[136] 表明，不完整 NTT 在某些情况下的性能优于完整 NTT 多项式乘法。对于 $n = 256 = 2^8$，完整 NTT 是指 8 层 NTT 多项式乘法。为了阐述不完整 NTT 性能优于完整 NTT 的原因，下文从所需算术指令的数量来分析和对比完整 8 层 NTT 以及 5/6/7 层不完整 NTT。

下文使用 M64 来表示两个 32 位整数相乘并得到 64 位乘积，A64 表示 64 位整数加法。在 SiFive E31 处理器上，正如本文 5.2.1 所述，M64 使用一条 `mul` 和一条 `mulh` 指令实现，其中每条指令的耗时为 5 个时钟周期，计算 A64 则需要使用 4 条基础指令。使用 MontMul 来表示一次蒙哥马利乘法，其需要使用一次 M64 操作和一次蒙哥马利约减（MontR）操作。比如，$c =$

$\mathsf{MontMul}(a,b)$ 用于计算 $c \equiv a \cdot b \bmod M$。蒙哥马利约减的开销可表示为 MontR = 1M64 + 1A32，其中 A32 用于表示所有的单周期指令，包括加法、减法和移位等指令。因此，一次 MontMul 的计算开销为 M64 + MontR = 2M64 + 1A32。下文使用 $<\mathsf{i},\mathsf{j}>$ 来表示 iM64 + jA32 的计算开销，其共计消耗 10i + j 个时钟周期。MontR/MontMul 的开销为 $<1,1>/<2,1>$。一次 CT 或 GS 蝴蝶变换的开销为 $\mathsf{butterfly} =<2,3>$，NTT/INTT 的每一层包含 $\frac{n}{2}$ 个蝴蝶变换，因此每一层的开销表示为 $\mathsf{layer} = \frac{\mathsf{n}}{2} <2,3>=<256,384>$。

8 层完整 NTT 中的逐点乘法的开销可直接表示为 $n\mathsf{MontMul} = n<2,1>=<512,256>$。

对于 7 层不完整 NTT 中的逐点乘法，其核心计算为 $c = c_0 + c_1 x = (a_0 + a_1 x) \cdot (b_0 + b_1 x) \bmod (X^2 - \gamma)$，其中 γ 为旋转因子常量。具体来说：

$$c_0 = \mathsf{MontMul}(a_0, b_0) + \mathsf{MontMul}(\mathsf{MontMul}(a_1, b_1), \gamma)$$
$$c_1 = \mathsf{MontR}(a_0 b_1 + a_1 b_0)$$

其中 c_1 的计算用到了延迟约减技术，因此 7 层 NTT 中逐点乘法的开销为

$$\frac{n}{2}(3\mathsf{MontMul} + 2\mathsf{M64} + 1\mathsf{MontR} + 1\mathsf{A64} + 1\mathsf{A32}) =<1152,1152>.$$

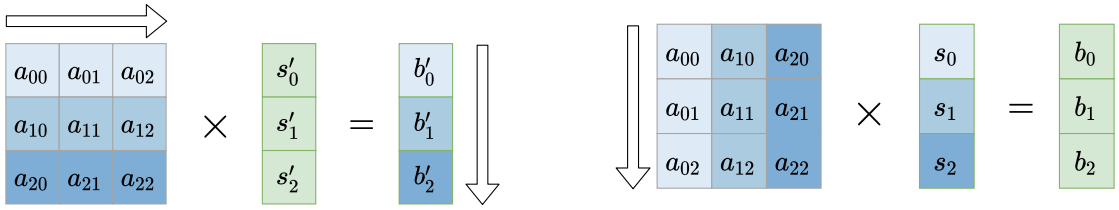
对于 6 层不完整 NTT 的逐点乘法，其核心计算为 $c = (a_0 + a_1 x + a_2 x^2 + a_3 x^3) \cdot (b_0 + b_1 x + b_2 x^2 + b_3 x^3) \bmod (X^4 - \gamma)$，整体的计算开销为 $\frac{n}{4}(3<8,12> + <5,13>) =<1856,3136>$。

对于 5 层不完整 NTT 的逐点乘法，其核心计算为两个 7 阶多项式的乘法运算，其整体开销为 $\frac{n}{8}(7<12,28> + <9,29>) =<2976,7200>$。

相比于 8 层完整 NTT，不完整 NTT 多项式乘法中 NTT 与 INTT 子程序的开销更低，但逐点乘法的开销更高。7 层不完整 NTT 中的逐点乘法相比于 8 层完整 NTT，增加的开销为 $<640,896>= 7296$ 个时钟周期，并且考虑的一层 NTT 运算的开销是 $<256,384>= 2944$ 个时钟周期。同时考虑到一个完整的多项式乘法需要计算 2 次 NTT 子程序、1 次 INTT 子程序和 1 次逐点乘法。因此相比于基于 8 层 NTT 的多项式乘法，基于 7 层 NTT 的多项式乘法能节省 1536 个时钟周期。同理可计算出 6 层 NTT 多项式乘法要快于 8 层 NTT，但慢于 7 层 NTT。上述分析旨在从算术指令数量的角度来解读不完整 NTT 性能优于完整 NTT 的原因。

5.5.3 NTT 优化实现技术

考虑到本节所选择的参数 $2^{16} < M < 2^{32}$，因此本节的实现属于 32 位 NTT，在 RV32IM 上无法使用改进版 Plantard 模乘算法，因此使用蒙哥马利模乘算法，其实现方式与算法 5.17 类似，区别在于本节没有对 $bq' \leftarrow bq^{-1} \bmod^{\pm} \beta$ 进行预计算，因此需额外消耗一条乘法指令，但能节省寄存器资源从而探索更多可能的层合并策略。



(a) $\mathbf{A} \cdot \mathbf{s}' = \mathbf{b}'$ (b) $\mathbf{A}^{\mathbf{T}} \cdot \mathbf{s} = \mathbf{b}$

图 5.2 Saber 中的矩阵向量乘法

在 30 个可用寄存器中，3 个寄存器用于函数传参、2 个寄存器用于保存蝴蝶变换中的临时值、2 个寄存器用于保存常量 M 和 M^{-1}，剩余 23 个寄存器用于循环控制和保存多项式系数以及旋转因子。

本工作给出了各种不同的 NTT 实现以对比其性能，具体包括 8 层完整 NTT、7/6/5 层不完整 NTT。在 8 层 NTT 实现中，层合并策略为 4+4，在 4 层合并实现中，需使用 1 个寄存器用于循环控制，16 个寄存器用于保存多项式系数，因此剩余 6 个寄存器用于加载旋转因子。下文用到的 3 层合并和 2 层合并也采用类似的思路，但具有更充裕的寄存器资源。7/6/5 层 NTT 实现采用的合并策略分别是 4+3、3+3 和 3+2。

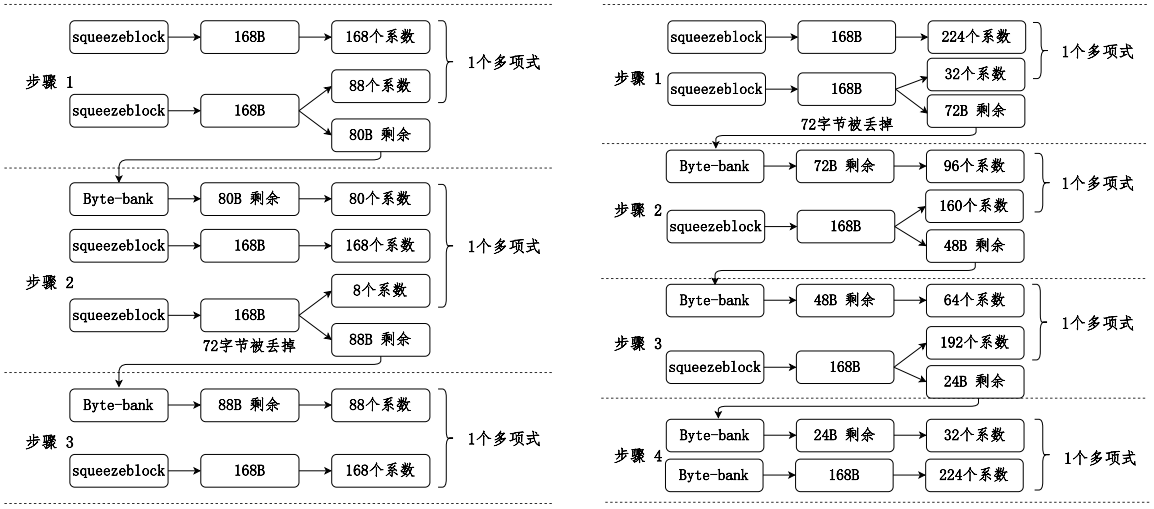
最终的性能测试表明，6 层 NTT 实现在 Saber 中具有最佳性能，更多的性能数据以及相关讨论将在本章 5.6 中给出。

5.5.4 内存优化技术

Saber 方案中的多项式矩阵 **A**、多项式向量 **s** 以及矩阵向量乘法需占用较多的内存资源。之前的相关工作[140] 为矩阵 **A** 提出了即时生成策略，称为 on-the-fly GenA 或 just-in-time GenA，从而降低 **A** 的内存占用。然而他们的 on-the-fly GenA 策略由于需要处理繁琐的剩余伪随机数，因此实现起来相当复杂。

本工作则提出了一种改进的 on-the-fly GenA 策略，并提出了 out-of-order GenA 策略，考虑到本文的改进与原始的 Saber 测试向量不兼容，因此我们将改进后的方案称为 Saber+ 方案。除此以外，本工作还为多项式向量 **s** 的生成提出了 on-the-fly GenS 策略。基于上述技术以及 **s** 的不同内存分配策略，本工作提出了三种不同的策略用于计算矩阵向量乘法，并分别应用到 LightSaber+、Saber+ 以及 FireSaber+ 方案中。

如图 5.2 所示，Saber 中共计出现两次矩阵向量乘法，分别是算法 2.17 中的 $\mathbf{A}^{\mathbf{T}} \cdot \mathbf{s}$ 和算法 2.19 中的 $\mathbf{A} \cdot \mathbf{s}'$。在 Saber 的参考实现中，矩阵 **A** 以行优先的形式生成，矩阵 $\mathbf{A}^{\mathbf{T}}$ 以列优先的形式生成。$\mathbf{A} \cdot \mathbf{s}'$ 中的核心运算是矩阵的第 i 行与向量的内积运算，内积的结果被保存到向量 $\mathbf{b}'$ 的第 i 个位置上。$\mathbf{A}^{\mathbf{T}} \cdot \mathbf{s}$ 的计算方法则有所不同，矩阵 $\mathbf{A}^{\mathbf{T}}$ 的第 i 列的每个多项式分别与向量 **s** 的第 i 个多项式相乘，所生成的 l 个多项式分别累加到 **b** 的第 i 个位置上。



(a) $l = 3$ (b) $l = 4$

图 5.3 on-the-fly GenS 技术示意图

on-the-fly GenA 技术 本工作所提出的 on-the-fly GenA 的基本原理是：仅当用到某多项式时再生成它，并且计算完成后复用其内存空间。以 $\mathbf{A} \cdot \mathbf{s}'$ 为例，其共计需要计算 l^2 个多项式乘法，其中每个 a_{ij} 只被用到一次。当使用 on-the-fly GenA 策略时，矩阵 $\mathbf{A}$ 的内存占用从 $l^2 \cdot 256 \cdot 2$ 字节降低到了 512 字节。该技术的副作用是需要维护 SHAKE128 的内部状态，并且需要处理剩余的伪随机数。

用于 $\mathbf{A} \cdot \mathbf{s}' = \mathbf{b}'$ 的 on-the-fly 技术 当计算 $\mathbf{A} \cdot \mathbf{s}' = \mathbf{b}'$ 时，可以直接将向量内积的计算结果转换为密文，因此无需保存整个多项式向量 $\mathbf{b}'$，只需占用一个多项式的内存空间。但考虑到 $\mathbf{s}'$ 中的每个多项式都被重复用到 l 次，如果每次用到 $\mathbf{s}'$ 中的多项式都重新生成的话，那么共计需要计算 $2l^2$ 次前向 NTT。如果保存 NTT 域的 $\hat{\boldsymbol{s}}'$，那么仅需计算 l^2+l 次前向 NTT，但需要占用 $l \cdot 256 \cdot 4$ 字节内存空间。因此理想的方式是能以列优先的形式生成矩阵 $\mathbf{A}$，因此只需保存一个 NTT 域的多项式 $\hat{\mathbf{s}}'_i$，而无需保存整个多项式向量。下文即将介绍的 out-of-order GenA 即可满足上述需求。

用于 $\mathbf{A}^{\mathbf{T}} \cdot \mathbf{s} = \mathbf{b}$ 的 on-the-fly 技术 当计算 $\mathbf{A}^{\mathbf{T}} \cdot \mathbf{s} = \mathbf{b}$ 时，on-the-fly GenS 使我们只需要保存一个多项式 $\mathbf{s}_i$，而无需保存整个向量 $\mathbf{s}$，但仍需保存整个 $\mathbf{b}$。具体来说，在计算矩阵向量乘法时，我们保存 NTT 域的 $\hat{\mathbf{s}}_i$，从而避免重复的前向 NTT 计算。

on-the-fly GenS 技术 多项式向量 $\mathbf{s}$ 由 $256 \times l$ 个系数组成，生成一个系数需要使用 μ 比特个伪随机数，对于 $l = \{2, 3, 4\}$，$\mu = \{10, 8, 6\}$，因此生成向量 $\mathbf{s}$ 分别需要 {640,768,768} 字节的伪随机数。考虑到一次 `keccak_squeezeblocks` 子程序调用能生成 168 字节伪随机数，因此分别需要调用 {4,5,5} 次来生成所需的伪随机数，并且会剩余 {32,72,72} 字节。本工作提出的优化思路为：（1）

对于 $l=2$，调用一次 keccak_squeezeblocks 子程序生成 168 字节的伪随机数，用于生成 128 个多项式系数，剩余的 8 字节伪随机数将被直接丢弃。最终调用 4 次 keccak_squeezeblocks 来生成多项式向量 **s**。（2）对于 $l=3$，第一个 168 字节伪随机数用于生成 168 个系数，第二个 168 字节的前 88 字节用于生成 88 个系数，从而构成了第一个完整的多项式，剩余的 80 字节则保存起来用于生成下一个多项式的 80 个系数。再次生成 2×168 字节伪随机数，生成完整的第二个多项式后，还剩余 160 字节，直接丢弃其中的 72 字节，保存剩余的 88 字节用于生成下一个多项式的 88 个系数。再生成 168 字节的伪随机数即可完成整个多项式向量的生成。上述过程见图 5.3a。（3）$l=4$ 的生成策略与 $l=3$ 类似，具体过程见图 5.3b。

on-the-fly GenA 技术 Karmakar 等人[140] 所提出的 on-the-fly GenA 技术涉及到复杂的逻辑以处理剩余的伪随机数。本工作所提出的 on-the-fly GenA 则简化了处理逻辑，并且不会增加 keccak 的调用次数。生成 **A** 中的一个多项式系数需要 13 比特的伪随机数，因此对于 $l=\{2,3,4\}$，分别共计需要 $\{1664,3744,6656\}$ 字节的伪随机数，分别需要调用 keccak_squeezeblocks() 的次数为 $\{10,23,40\}$ 次。考虑到每 2 个多项式需要 5×168 字节的伪随机数，即每个多项式平均需要 2.5×168 字节。基于上述观察，我们所设计的 on-the-fly GenA 策略为：先用 2×168 字节的伪随机数生成 206 个多项式系数；然后使用第三个 168 字节中的 82 字节生成 50 个系数以构成完整的第一个多项式，并且还有 86 字节的剩余；对于 86 字节的剩余，我们只保留其中的 82 字节用于生成下一个多项式的 50 个系数。然后再使用 2×168 字节的伪随机数生成 206 个多项式系数以构成完整的另一个多项式。总之，本工作所设计的 on-the-fly GenA 技术的基本原理是调用 5 次 keccak_squeezeblocks()，利用所得到的 5×168 字节的伪随机数生成 2 个多项式。

out-of-order GenA 技术 在 Kyber 方案中，矩阵 **A** 中的每一个项 a_{ij} 的生成没有顺序限制，称这种方法为 out-of-order GenA。在 Kyber 中，为了生成 a_{ij}，keccak_absorb() 子程序的输入包括 seed、i 和 j，生成 l^2 个多项式需要调用 l^2 次 keccak_absorb()。但在 Saber 中，keccak_absorb() 只被调用了 1 次，因此矩阵 **A** 只能以行优先的顺序生成，矩阵 $\mathbf{A^T}$ 只能以列优先的顺序生成，称这种方法为 in-order GenA。

out-of-order GenA 的优势是生成顺序没有限制，劣势是需调用 l^2 次 keccak_absorb()，并且每次构造完某个 a_{ij} 后便直接丢弃掉剩余的伪随机数，会导致 keccak_squeezeblocks() 调用次数的增加。in-order GenA 的性能更好，但生成顺序受限。

在本工作的实现中，out-of-order GenA 技术与 on-the-fly 策略组合使用。对于 Saber 的三个安全参数，当使用 out-of-order GenA 技术生成矩阵 **A** 时，相比于 in-order GenA，keccak_absorb() 调用次数增加了 {3,8,15} 次，keccak_squeezeblocks() 调用次数增加了 {2,4,8} 次。

计算 $\mathbf{A^T}\cdot\mathbf{s}$ 时，on-the-fly GenA 与 on-the-fly GenS 技术可以结合使用，前向 NTT 的计算次

表 5.2　C908 处理器 RV32IM、RV64IM 和 RVV 上，ML-KEM 的 NTT、INTT 与逐点乘法的时钟周期对比。RV32IM 和 RV64IM 上的实现使用本章所设计的改进版 Plantard 模乘算法，RVV 上的实现使用蒙哥马利模乘算法。NTT 和 INTT 分别使用 CT 和 GS 蝴蝶变换实现，RV32IM 和 RV64IM 上层合并策略分别为 4+3 和 3+4，RVV 上层合并策略分别为 1+6 和 6+1

实现	NTT	INTT	逐点乘法
RV32IM 上的参考实现	34104	54925	17690
RV64IM 上的参考实现	34978	55719	17817
本工作单发射 RV32IM 上的性能优化版本	13218	11427	2891/3916/5900
本工作双发射 RV32IM 上的实现	5714	6005	1673/1962/2668
本工作双发射 RV64IM 上的实现	6609	6996	2122/2507/3245
本工作 RVV 上的实现	1575	1840	753

数是 l^2+l 次。然而计算 $\mathbf{A}\cdot\mathbf{s}'$ 时，如果使用 on-the-fly GenS 来减少向量 $\mathbf{s}'$ 的内存占用，前向 NTT 的计算次数是 $2l^2$ 次。基于上述观察，本工作提出了计算矩阵向量乘法的三种不同的策略：（1）策略 1：保存整个向量 $\mathbf{s}'$，前向 NTT 的计算次数是 $2l^2$ 次。（2）策略 2：保存 NTT 域的 $\hat{\mathbf{s}}'$，前向 NTT 的计算次数是 l^2+l 次。（3）策略 3：使用 out-of-order GenA 技术以列优先的顺序生成矩阵 $\mathbf{A}$，结合 on-the-fly GenS 技术，我们只需要保存一个 NTT 域的 $\hat{\mathbf{s}}_\mathbf{i}$，前向 NTT 的计算次数是 l^2+l 次。上述三个策略中，逐点乘法与 INTT 的计算次数均为 l^2 次。

简言之，策略 1 的内存占用最少但性能最差，策略 2 性能最佳但内存占用最大，策略 3 的内存占用与性能均介于前两者之间。为了能使 Saber 的三个安全参数均能部署到内存受限的嵌入式设备上，本工作分别将策略 1、2 和 3 应用于 FireSaber+、LightSaber+ 和 Saber+。

5.6　性能评估

考虑到本文第六章将对 ML-KEM 和 ML-DSA 方案的另一个耗时运算 SHA-3 进行优化，因此本节只报告 NTT 相关函数的性能，ML-KEM 和 ML-DSA 方案的整体性能将在第六章给出。用于性能评估的处理器包括单发射 RV32IM、双发射 RV32IM、双发射 RV64IM 和 RVV 处理器，均在本章 5.2.1 中给出了介绍。如无特殊说明，下文中的时钟周期数据均是通过多次运行对应程序取均值或中间值得到。考虑到实现逐点乘法时使用了非对称乘法技术，逐点乘法往往有多个变体函数，因此下文表格中逐点乘法对应的列往往有多项数据。

表 5.3　C908 处理器 RV32IM、RV64IM 和 RVV 上，ML-DSA $\mathbb{Z}_{8380417}[X]/(X^n+1)$ 上 NTT、INTT 与逐点乘法的时钟周期对比。RV64IM 上的实现使用改进版 Plantard 模乘算法，RV32IM 和 RVV 上的实现使用蒙哥马利模乘算法。NTT 和 INTT 分别使用 CT 和 GS 蝴蝶变换实现，RV32IM 上层合并策略分别为 3+3+2 和 2+3+3，RV64IM 和 RVV 上层合并策略分别为 4+4 和 4+4

实现	NTT	INTT	逐点乘法
RV32IM 上的参考实现	34236	43837	7454
RV64IM 上的参考实现	26485	31602	5661
本工作单发射 RV32IM 上的实现	13223	11114	2856
本工作双发射 RV32IM 上的实现	7054	7561	2026
本工作双发射 RV64IM 上的实现	8258	8484	2320
本工作 RVV 上的实现	3395	3540	759

5.6.1　ML-KEM 方案的 NTT

双发射 RV32IM、双发射 RV64IM 和 RVV 处理器　所选择的开发板是嘉楠 K230，其配备了平头哥玄铁 C908 双发射 RISC-V 处理器，支持 RV{32,64}GCBV 指令集，此处仅关注其中的 RV32IM、RV64IM 和 RVV 指令集。所使用的 GCC 版本为 `Xuantie-900 linux-5.10.4 glibc gcc toolchain v2.8.0`，所有程序均使用`-O3`选项编译。表 5.2 给出了实验数据以及相关对比：

- 本章双发射 RV32IM 上的 NTT 相关函数实现速度是单发射实现的 1.7 倍 ~ 2.3 倍，是 RV32IM 上参考实现的 3.0 倍 ~ 9.1 倍。
- 本章双发射 RV64IM 上的 NTT 相关函数实现速度是参考实现的 5.3 倍 ~ 8.4 倍。RV64IM 实现略慢于 RV32IM 实现的主要原因是 RV64IM 上的乘法指令 `mul{h}` 比 RV32IM 上的乘法指令慢，见表 5.1。
- 本章 RVV 上的 NTT 相关函数实现速度是本章 RV32IM 上实现的 2.2 倍 ~ 3.6 倍，是 RV32IM 上参考实现的 21.7 倍 ~ 29.9 倍。

5.6.2　ML-DSA 方案的 NTT

双发射 RV32IM、双发射 RV64IM 和 RVV 处理器　表 5.3 给出了双发射 RV32IM、双发射 RV64IM 和 RVV 处理器上，ML-DSA $\mathbb{Z}_{8380417}[X]/(X^n+1)$ 上的 NTT、INTT 与逐点乘法的时钟周期对比：

- 本章双发射 RV32IM 上的 NTT 相关函数速度是单发射实现的 1.4 倍 ~ 1.9 倍，是 RV32IM 上参考实现的 3.7 倍 ~ 5.8 倍。
- 本章双发射 RV64IM 上的 NTT 相关函数速度是 RV64IM 上参考实现的 2.4 倍 ~ 3.7 倍。

表 5.4　SiFive E31 处理器上，Saber 方案的性能对比，单位为 k 时钟周期，$k = 1000$。KeyGen、Encaps 与 Decaps 分别表示密钥生成、封装与解封装。Karmakar 等人仅提供了 Saber 的性能数据[140]。

方案	实现	KeyGen	Encaps	Decaps
LightSaber	本工作 C 实现	2219	2202	2030
	本工作汇编实现	2560	2205	1773
Saber	Karmakar 等人的 M0-mem 实现	6546	6614	7133
	本工作 C 实现	3647	3881	3755
	本工作汇编实现	3809	3594	3193
	加速比	42%	46%	55%
FireSaber	本工作 C 实现	5056	6076	6019
	本工作汇编实现	5063	5360	4928

表 5.5　SiFive E31 处理器上，Saber 方案的内存占比对比，单位为字节。KeyGen、Encaps 与 Decaps 分别表示密钥生成、封装与解封装。Karmakar 等人仅提供了 Saber 的内存占比数据[140]。

方案	实现	KeyGen	Encaps	Decaps
LightSaber	本工作 C 实现	3812	4404	4436
	本工作汇编实现	3804	4396	4428
Saber	Karmakar 等人的 M0-mem 实现	4984	4696	5800
	本工作 C 实现	4324	4936	4948
	本工作汇编实现	4316	4924	4940
	加速比	+13%	-5%	+15%
FireSaber	本工作 C 实现	4884	4916	4948
	本工作汇编实现	4876	4908	4940

- 本章 RVV 上的 NTT 相关函数速度是本章 RV32IM 上实现的 2.1 倍 ~ 2.7 倍，是 RV32IM 上参考实现的 9.8 倍 ~ 12.4 倍，是本章 RV64IM 上实现的 2.4 倍 ~ 3.1 倍，是 RV64IM 上参考实现的 7.5 倍 ~ 8.9 倍。

5.6.3　Saber 方案

所选择的开发板是 SiFive E310，其配备了 32 位单发射 E31 RISC-V 处理器，支持 RV32IMAC 指令集。所使用的 GCC 版本为 `SiFive GCC 8.3.0`，编译选项为`-Os`。`-O3` 是用于获得最佳性能的常用选项，但对于 Saber 在 SiFive E31 处理器上的实现，`-Os` 选项所得到的性能更优。原因是 E31 核心提供了 8KB 的 ITIM（Instruction Tightly Integrated Memory），从 ITIM 中获取指令性能更优。当使用`-O3` 编译程序时，所得到的可执行文件无法被全部放置到 ITIM 中，部分程序会被放置到 ROM 中，从 ROM 中获取指令的时延更高。因此，`-Os` 选项编译得到的尺寸更小的可执行文件对应的性能更优。

性能对比 由于 SiFive E31 处理器仅提供了 16KB 的内存，Saber 参考实现因内存占用过大无法运行。Karmakar 等人[140] 所提供的 M0-mem 版本实现提供了一系列内存优化技术，但其仅提供了 Saber 的性能与内存数据。由表 5.4 可知，本工作的汇编实现相比于 Karmakar 等人的实现，KeyGen、Encaps 和 Decaps 的性能提升分别为 42%、46% 和 55%。

内存对比 表 5.5 给出了本工作实现与 Karmakar 等人实现的内存占用对比，本工作汇编实现的 KeyGen 与 Decaps 的内存占用均比 Karmakar 等人的实现低，分别降低了 13% 和 15%。值得注意的是，以 Saber 为例，本工作实现的内存占用均低于 5KB，对于 Saber 方案在资源严重受限的物联网设备上的部署有一定的启发意义。

5.7 可扩展性与安全性讨论

本章主要研究 ML-KEM、ML-DSA 和 Saber 的 NTT 多项式乘法在各种 RISC-V 处理器上的优化实现，所涵盖的 RISC-V 处理器包括双发射 RV32IM、双发射 RV64IM 和 RVV，得益于改进版 Plantard 模乘算法的性能优势以及本章所提出的双发射流水线优化技术，本章的各种实现均刷新了性能记录。本章所使用的改进版 Plantard 模乘也可应用于其它格密码方案，例如模数小于 2^{16} 的 NewHope[120] 和 NTTRU[136] 方案。此外，本章的研究也可应用于我国国产自主可控的后量子密码方案，如复旦大学赵运磊教授团队设计的：基于 NTRU 的密钥封装机制 CTRU 和 CNTR 方案[141]、基于模格的 AKCN-MLWE 方案；中科院信息工程研究所路献辉教授团队设计的 LAC 方案[142]。这些方案的模数均小于 2^{16}，且都能使用 NTT 加速多项式乘法运算，因此这些方案在嵌入式处理器上的高效实现均可采用本章的性能优化方法进行加速。

本章所使用的改进版 Plantard 模乘的优化实现思路可以迁移至其他具有类似指令的处理器上，从而能取代蒙哥马利模乘和 Barrett 模乘来加速格密码性能。比如，ARM Cortex-M7 处理器上的格密码优化实现可参考 ARM Cortex-M4 处理器上的优化实现思路，因为该处理器也支持类似的 SIMD 指令[143]。

本章实现中所有私钥相关的计算均为恒定执行时间，并避免了私钥相关的分支操作和内存访问。值得注意的是，ARM Cortex-M3 处理器上的 32×32 位乘法指令是非恒定执行时间的，因此在实现 ML-DSA 密钥生成和签名时，考虑到这两个算法均需要处理私钥相关的计算，Huang 等人[133] 使用多模数 NTT 相关技术将其中的 32 位 NTT 转换为了 16 位 NTT，从而避免使用这些非恒定时间的指令。

5.8 本章小结

针对格密码中多项式乘法耗时占比高的问题，本章研究了国际后量子密码标准 ML-KEM 和 ML-DSA 以及 Saber 方案的 NTT 多项式乘法在各种 RISC-V 处理器上的性能优化，本章的主要

优化点是整数模乘和 NTT 多项式乘法。改进版 Plantard 模乘在各种处理器上的性能优于目前广泛使用的蒙哥马利模乘与 Barrett 模乘，更大的输入范围与更小的输出范围使得在 NTT/INTT 实现过程中可以设计出更优的延迟约减策略，从而获得更快的性能表现。最终，本章在各种 RISC-V 上的优化实现均刷新了性能记录，性能最多是之前实现的 29.9 倍。本章的研究可以有效促进格密码在各种面向物联网场景的嵌入式处理器上的应用和普及，并对我国国产自主可控的后量子密码在嵌入式平台上的高效实现具有一定的启发作用。

第六章　SHA-3 性能优化研究及其在格密码优化中的应用

SHA-3（Secure Hash Algorithm 3）是由美国国家标准与技术研究院（NIST）于 2015 年发布的加密哈希函数标准，其核心运算为 Keccak-f1600 算法。NIST 制定的后量子密码标准 ML-KEM 与 ML-DSA 方案均依赖于 SHA-3 算法，其中多项式矩阵和多项式向量生成过程中的耗时运算是 SHA-3 标准中的扩展输出函数，即 SHAKE128 与 SHAKE256，这些运算成为了 ML-KEM 与 ML-DSA 方案的性能瓶颈之一。除此以外，RISC-V 架构的模块化设计原则也带来了指令集组合碎片化的问题，SHA-3 在各种不同指令集组合上的优化实现研究仍不够充分。为了解决上述问题，本章研究了各种 RISC-V 处理器上的 Keccak-f1600 性能优化，涵盖的 RISC-V 指令集包括 8 种常见组合，即 RV{32,64}I{B}{V}。本章详细回顾了 Keccak-f1600 的各种优化实现技术，并针对不同的嵌入式处理器构造了最优的技术组合。实验结果表明，本章的研究刷新了 Keccak-f1600 在各种嵌入式处理器上的性能记录，性能提升最高可达之前实现的 4 倍。此外，本章还研究了如何使用向量指令（RVV）来优化 Keccak-f1600，并论证了标量-向量混合实现技术在各种指令集组合上的可行性。将第五章优化的 NTT 实现与本章优化的 Keccak-f1600 实现集成到 ML-KEM 与 ML-DSA 方案中后，不仅刷新了 ML-KEM 与 ML-DSA 方案的性能记录，还进一步丰富了格密码在 RISC-V 架构上的软件生态。

6.1　引言

SHA-3 是由 Keccak 算法衍生出来的一个加密哈希函数标准，属于 NIST 发布的安全哈希系列中的第三个成员。SHA-3 于 2015 年被正式发布，作为 SHA-2 的补充和替代方案，SHA-3 并非 SHA-2 的直接改进，而是采用了完全不同的结构和设计理念。SHA-3 基于 Keccak 算法，它采用了一种称为“海绵构造”（Sponge Construction）的设计。这种构造方式使得 SHA-3 具有较强的安全性和灵活性。海绵构造的基本思想是通过一个“吸收”阶段和一个“输出”阶段对输入数据进行处理。在吸收阶段，输入的数据被逐块吸收并与内部状态进行混合，而在输出阶段，通过输出函数将最终的结果提取出来。这种结构的最大优点是可以处理任意长度的输入数据，并且能够生成不同长度的输出值。

Keccak 算法的设计目标是抵抗各种已知的加密攻击，例如差分攻击和线性攻击。Keccak 采用了一种不同于传统 Merkle-Damgård 结构的迭代式设计，使其在安全性上具有更好的保证。Keccak 的核心是一个基于置换网络的操作，其中涉及到的基本操作包括异或、置换和旋转。这些操作确保了信息在加密过程中能够充分混合，从而增强了算法的抗攻击能力。与 SHA-2 不同，SHA-3 在性能上并未表现出明显的提升，但它的设计更加注重安全性和灵活性，特别适用于对

抗量子计算威胁的环境。SHA-3 可以用于多种应用场景，包括数据完整性校验、数字签名、密钥生成等。

总体而言，SHA-3 和 Keccak 代表了当前加密哈希函数领域的一种新趋势，通过创新的构造和严密的数学基础，提供了一种更加安全且具备扩展性的哈希算法。随着量子计算技术的发展，SHA-3 的设计理念有可能为未来的加密标准提供有益的启示。

本章贡献 Keccak-f1600 置换算法是 SHA-3 的核心组件，SHA-3 是格密码 ML-KEM 与 ML-DSA 方案的性能瓶颈之一，为了解决这一问题，本章研究了 Keccak-f1600 在 8 种不同的 RISC-V 指令集组合上的优化实现，并重新审视了各种优化实现技术，从而为各种指令集构造最优技术组合，具体地：

- 对于 RV64I 上的实现，较大的寄存器分配压力加大了双发射流水线优化难度，为此本章设计了专门的流水线优化方案来改进 Keccak-f1600 的性能，并证明了通过少量的栈访问来换取更优的流水线执行效率能改进程序性能。
- 对于 RV64IB 上的实现，本章阐述了如何利用 B 扩展提供的 `rori` 和 `andn` 指令来加速 Keccak-f1600 ，这两条指令不仅大幅度减少了 Keccak-f1600 计算所需的指令数，还降低了流水线优化难度。
- RV32I 与 RV32IB 上的实现难点在于没有足够的寄存器来容纳 Keccak-f1600 的 1600 比特的状态，因此不得不使用大量的内存访问指令来读写 Keccak-f1600 的状态，这又导致大量的 “加载使用数据冒险” 从而导致性能降级。本章以流水线友好性为主要目标，设计了专门的寄存器分配方案和双发射流水线优化方法。
- 对于 RVV（RISC-V Vector）上的实现，本章不仅阐述了如何利用向量指令实现 Keccak-f1600 ，还探索了标量-向量混合实现技术，论证了混合实现技术在各种指令集组合上的可行性。
- 在平头哥玄铁 C908 处理器上的测试表明，相比于之前的最优实现，本章在 RV32I、RV32IB、RV64I 和 RV64IB 上的 Keccak-f1600 实现性能提升分别为 12%、98%、90% 和 36%。对于标量-向量混合实现，本章的 RV32IVx3 与 RV32IBVx3 实现的性能提升分别为 98% 和 77%。
- 最终，本章还将本文第五章优化的 NTT 实现与本章优化的 Keccak-f1600 实现集成到了 ML-KEM 与 ML-DSA 方案中，并给出了性能测试，表明本文的 ML-KEM 与 ML-DSA 实现在各种 RISC-V 指令集组合上均刷新了性能记录，性能分别最高是之前实现的 4.7 倍和 4.2 倍。

算法 6.1: Keccak-f1600 的一轮

输入: 1600 比特的状态 A；常量 RC

输出: 1600 比特的状态 A

θ step:

$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad \forall x \text{ in } 0 \ldots 4$

$D[x] = C[x-1] \oplus \mathtt{ROT}(C[x+1],1), \quad \forall x \text{ in } 0 \ldots 4$

$A[x,y] = A[x,y] \oplus D[x], \quad \forall (x,y) \text{ in } (0 \ldots 4, 0 \ldots 4)$

ρ and π steps:

$B[y,2x+3y] = \mathtt{ROT}(A[x,y], r[x,y]), \quad \forall (x,y) \text{ in } (0 \ldots 4, 0 \ldots 4)$

χ step:

$A[x,y] = B[x,y] \oplus ((\mathtt{NOT}\ B[x+1,y])\ \mathtt{AND}\ B[x+2,y]), \quad \forall (x,y) \text{ in } (0 \ldots 4, 0 \ldots 4)$

ι step:

$A[0,0] = A[0,0] \oplus \mathrm{RC}$

return A

6.2 Keccak-f1600 优化技术介绍

SHA-3 家族包含 4 个哈希函数，即 SHA3-{224, 256, 384, 512}，其中数字表示输出摘要的比特长度；还包含 2 个扩展输出函数，即 SHAKE{128, 256}，其中数字表示安全强度。SHA-3 家族中的每个函数均基于 Keccak-f1600 置换算法构造，Keccak-f1600 由 24 轮组成，每一轮包含 5 个变换，即 $\theta, \rho, \pi, \chi, \iota$。算法 6.1 给出了 Keccak-f1600 一轮的计算流程，其中 $r[x,y]$ 与 RC 均为固定的常量，$\oplus$ 表示异或 XOR 操作，ROT 表示 64 位左旋转操作。

Keccak-f1600 的状态长度为 1600 比特，可被划分为 3 个维度，分别由 $x, y \in \mathbb{Z}_5$ 和 $z \in \mathbb{Z}_{64}$ 表示，比特索引 $i = z + 64(5y + x)$ 与坐标 (x, y, z) 对应。1600 比特状态也可被视为 5×5 维的矩阵 $A[x,y]$，其中每个元素为 64 比特值。文献[144] 给出了实现 Keccak-f1600 的各种优化技术，下文将简述这些优化技术。

位交错（bit interleaving） 该技术主要用于在 32 位处理器上实现 Keccak-f1600，通过改变状态的编码将 64 位旋转操作转换为 32 位旋转操作。具体地，一个 64 比特状态 $L[z] = A[x,y,z]$ 被映射为两个 32 位状态 U_0 和 U_1，其中 $U_0[j] = L[2j]$、$U_1[j] = L[2j+1]$、$j \in [0, 31]$。将 L 旋转 2τ 比特等价于分别旋转两个 32 位状态 τ 比特。将 L 旋转 $2\tau+1$ 比特等价于 $U_0 \leftarrow \mathtt{ROT}_{32}(U_1, \tau+1)$、$U_1 \leftarrow \mathtt{ROT}_{32}(U_0, \tau)$。当旋转偏移量为 1 时，只需执行 1 个 32 位旋转，Keccak-f1600 中的 29 个 64 位旋转中的 6 个符合这一情况。本节参考 XKCP 开源代码库中的编解码方法[1]来分析该

[1]https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/plain-32bits-inplace/KeccakP-1600-inplace32BI.c中的 `toBitInterleavingAndAND` 和 `fromBitInterleaving` 宏定义

技术所引入的开销。将两个 32 位值进行编码并与对应的 Keccak-f1600 状态进行异或消耗 56 条逻辑指令，对于 ML-KEM 和 ML-DSA，该过程大致需要 $56 \times 5 = 280$ 条指令。对 2 个 32 位状态解码消耗 48 条逻辑指令，对于 SHA3-{256,512} 和 SHAKE{128,256}，解码开销分别为 $\{17, 9, 21, 17\} \times 48 = \{816, 432, 1008, 816\}$ 条指令，编解码一般在 SHA-3 或 SHAKE 层进行。

通道取补（lane complementing） 该技术通过将部分状态取补，从而减少 χ 步骤中的 `NOT` 操作数量，并且将部分 `AND` 操作转换为了 `OR` 操作。该技术对于不支持 `a AND NOT(b)` 指令的架构是有益的，比如本章 RV64IM 上的 Keccak-f1600 实现采用了该技术，通过在保存状态前和加载状态后分别对 6 个 64 位状态执行 `NOT` 操作，将 χ 步骤中的 `NOT` 操作数量从 25 降低到了 8。

就地实现（in-place implementation） 如算法 6.1 所述，ρ 和 π 步骤对 25 个 64 位状态执行旋转操作，并将结果保存至临时的 $B[]$ 中；χ 步骤则以 $B[]$ 为输入，并将结果保存至 $A[]$ 中。就地实现计算旨在使 $B[]$ 和 $A[]$ 的存储位置相同，从而保证 n 轮计算结束后 $A[x, y]$ 的存储位置保持不变。文献[144] 给出了一种 $n = 4$ 的就地实现方法，该方法需要在实现时展开 4 轮计算。Becker 等人[145] 则提出了一种 $n = 1$ 的就地实现方法，称为单轮就地实现，本章便采用该方法。

延迟旋转（lazy rotation） Becker 等人[145] 为 ARMv8-A 架构上的 Keccak-f1600 实现提出了延迟旋转技术，利用了 barrel 移位特性消除了 Keccak-f1600 计算过程中的大部分旋转操作。但该方法只能用于 ARMv8-A 或 ARMv7-M 架构，无法应用于 RISC-V，因为 RISC-V 指令集不支持 barrel 移位特性。

ARM Cortex-M3 与 Cortex-M4 处理器上的 Keccak-f1600 优化 Huang 等人[133] 在 ARMv7-M 上对 Keccak-f1600 进行了性能优化，应用了两项优化技术：

1. 延迟旋转：XKCP 中的实现[1]使用 `ror` 指令来计算 ρ 步骤的旋转操作，这一实现方法的可读性较好，但每一轮需要使用 47 条 `ror` 指令。Becker 等人[145] 为 ARMv8-A 架构上的 Keccak-f1600 优化实现所提出的延迟旋转技术可以消除大部分旋转指令，Huang 等人将这一技术迁移至 ARMv7-M 架构上。该技术的基本原理为：将 ρ 步骤的旋转操作推迟到下一轮的 θ 步骤中执行，得益于 barrel 移位特性，下一轮的 θ 步骤并不会引入额外的计算开销，因此可以提升 Keccak-f1600 的计算效率。这一技术将每轮计算所需的 `ror` 指令数从 47 降低到了 3。考虑到该方法需要将多轮计算展开从而才能将旋转操作推迟到下一轮，这会增加可执行文件的大小，因此 Huang 等人实现了两个不同的版本。性能优化版本对 Keccak-f1600 的所有轮次都使用延迟旋转技术，这需要重写首轮和尾轮代码来构造延迟旋转所需的流程。考虑到使用了文献[144] 中的 $n = 4$ 的就地实现方法，对于

[1]https://github.com/XKCP/XKCP/commit/7fa59c0ec4b5802b7c269ddd9ef0ef35999b4f0f

每 4 轮计算，首轮和尾轮均有 2 个不同的变体，因此最终可执行文件增大了 50%。空间优化版本只对 $\frac{3}{4}$ 的轮次使用延迟旋转技术，因此不会使可执行文件变大，其速度会稍微慢于性能优化版本。

2. 内存访问流水线优化：XKCP 开源代码库中 ARMv7-M 上的 Keccak-f1600 实现中的 5 条 `ldr` 指令中只有 2 条是连续的，见代码片段 6.1。在 ARM Cortex-M3 和 Cortex-M4 上，将这 5 条 `ldr` 指令全部组合起来可以节省 3 个时钟周期。Huang 等人通过稍微调整寄存器分配方案将上述 5 条 `ldr` 指令全部组合了起来，并且尽可能地将 `str` 指令放置到 `ldr` 指令之后来减少开销，见代码片段 6.2。

```
1  .macro xor5   result,b,g,k,m,s
2    ldr     \result, [r0, #\b]
3    ldr     r1, [r0, #\g]
4    eors    \result, \result, r1
5    ldr     r1, [r0, #\k]
6    eors    \result, \result, r1
7    ldr     r1, [r0, #\m]
8    eors    \result, \result, r1
9    ldr     r1, [r0, #\s]
10   eors    \result, \result, r1
11 .endm
```

代码片段 6.1　XKCP 中的 ARMv7-M 汇编代码片段

```
1  .macro xor5  result,b,g,k,m,s
2    ldr    \result, [r0, #\b]
3    ldr    r1,  [r0, #\g]
4    ldr    r5,  [r0, #\k]
5    ldr    r11, [r0, #\m]
6    ldr    r12, [r0, #\s]
7    eors   \result, \result, r1
8    eors   \result, \result, r5
9    eors   \result, \result, r11
10   eors   \result, \result, r12
11 .endm
```

代码片段 6.2　Huang 等人的 ARMv7-M 汇编代码片段

6.3　RISC-V 上的 Keccak-f1600 优化

本节给出 RISC-V 架构上的 Keccak-f1600 优化实现，所涵盖的指令集组合包括 RV64I、RV64IB、RV32I、RV32IB 和 RVV，所使用的处理器型号是平头哥 C908。除此以外，本节还论证了标量-向量混合实现技术的可行性，其基本思想是将标量实现与向量实现交替混合执行，本节的研究证明，在 C908 处理器上，该技术对于 RV32IV 和 RV32IBV 指令集上的 Keccak-f1600 实现可以带来明显的性能改进。考虑到本章所使用的玄铁 C908 是一款双发射处理器，因此本节还重点研究如何针对双发射 CPU 进行流水线优化。

6.3.1　RV64I 和 RV64IB 上的优化实现

RV64I 上的 64 位左旋转操作　如算法 6.1 所示，θ 和 $\rho\pi$ 步骤需要执行 64 位左旋转操作。由于 RV64I 指令集没有提供旋转指令，因此我们使用指令序列 `slli t,a,n; srli b,a,64-n; xor`

`b,b,t` 来实现 `b ← ROT(a,n)`。考虑到一轮 Keccak-f1600 计算中的大多数中间值不能被直接覆盖，因此上述指令序列一般需要占用 2 个临时寄存器，分别用于保存中间值和计算结果。

RV64I 上的通道取补技术 RV64I 没有提供计算 `a AND NOT(b)` 的指令，因此用于减少 `NOT` 操作的通道取补技术对于 RV64I 上的 Keccak-f1600 实现是有益的。通道取补技术需要分别在加载所有状态后和保存所有状态前对 25 个 64 位状态中的 6 个执行 `NOT` 操作，每一轮计算中 χ 步骤的 `NOT` 操作数量从 25 减少到了 8，同时部分 `AND` 操作被转换为了 `OR` 操作。

RV64IB 上的 rori 和 andn 指令 B 扩展提供了 64 比特旋转指令 `rori`，可直接用于计算算法 6.1 中的 `ROT` 操作；B 扩展还提供了 `andn` 指令，可直接用于计算 χ 步骤的 (`NOT` $B[x+1,y]$) `AND` $B[x+2,y]$。一方面，相比于 RV64I 上用于计算旋转的 3 指令序列，使用 `rori` 指令可以降低寄存器分配压力并且可以减少 RAW（Read After Write）数据冒险。另一方面，得益于 B 扩展所提供的 `andn` 指令，RV64IB 上的 Keccak-f1600 实现将不再使用通道取补技术。

寄存器分配和就地实现 RISC-V 架构提供了 30 个编程可用的寄存器，因此我们使用 25 个寄存器来保存 Keccak-f1600 的 1600 比特状态，剩余 5 个寄存器可用于临时计算。本节采用 Becker 等人[145] 所设计的就地实现方法，其基本思路为：算法 6.1 中的 $\rho\pi$ 步骤将计算结果保存至 $B[]$ 中，我们使用 `loc` X 来表示 X 所占用的寄存器。就地实现的目标是在计算 $\rho\pi$ 时，使 `loc` $B[]$ 稍微偏移 `loc` $A[]$，计算 χ 时再将 $A[]$ 移回原来的位置。具体地，对于 $x \notin \{0,1\}$，使得 `loc` $B[x,y]$ = `loc` $A[x,y]$；对于 $x \in \{0,1\}$ 且 $y \in \{1,2,3,4\}$，使得 `loc` $B[x,y]$ = `loc` $A[x,(y+1)\%5]$；使用额外的寄存器来保存 $B[0,0]$ 和 $B[1,0]$。该方法相比于文献[144] 中的 4 轮展开就地实现方法的优势在于无需展开多轮计算，既有利于简化编程也能减少可执行文件大小。

RV64I 上的双发射优化 相比于 Becker 等人在 ARMv8-A 上的实现[145]，本节的双发射优化更加复杂，具体原因为：(1) ARMv8-A 架构拥有 31 个可用寄存器，而 RV64 只有 30 个；(2) ARMv8-A 提供的 `bic` 指令可直接计算 `c ← a AND NOT(b)`，而 RV64I 则没有类似的指令；(3) Becker 等人的实现利用了 ARMv8-A 架构提供的 barrel 移位特性消除了大部分旋转指令，这使得他们有更多的寄存器用于流水线优化。RV64I 上则需要使用 3 条指令来实现旋转操作，并且需要占用一个寄存器用于存储中间值，这也加大了寄存器分配压力。而且这 3 条指令存在 RAW 数据冒险，会导致流水线停顿。如果我们直接采用 Becker 等人的寄存器分配方案，使用五个寄存器保存 θ 步骤产生的 $C[]$，那么将没有剩余寄存器来计算 $D[]$ 了，从而不得不使用额外的内存访问指令。

考虑到 θ 步骤是一轮计算中寄存器分配压力最大的步骤，因此下文将重点讲解这一步骤的寄存器分配方案，具体的计算顺序为：

1. $C[0], C[2] \rightarrow D[1]$。该计算序列使用 3 个临时寄存器，并且可以充分地进行指令交替来避免流水线停顿。
2. $C[1] \rightarrow A[1,*] \oplus D[1] \rightarrow C[4]$。这一计算序列同样可以避免流水线停顿，计算完成后 $D[1]$ 所占用的寄存器可以被释放，此时 $C[0], C[1], C[2], C[4]$ 占用了 4 个寄存器，仅剩一个可用。
3. $D[3] \rightarrow C[3] \rightarrow A[3,*] \oplus D[3]$。$D[3]$ 的计算使用了仅剩的 1 个临时寄存器，并且后续的计算等待 $C[2]$ 所占用寄存器的释放，因此用于计算 $D[3]$ 的两个移位和两个 `XOR` 操作无法进行充分地交替来消除 RAW 数据冒险。因此，我们决定将一个暂时用不到的 64 位状态暂存到栈里来获取一个临时寄存器，用于充分地指令交替，从而消除上述 RAW 数据冒险。
4. $D[4] \rightarrow A[4,*] \oplus D[4] \rightarrow D[2] \rightarrow A[2,*] \oplus D[2] \rightarrow D[0] \rightarrow A[0,*] \oplus D[0]$。借助于上述通过栈访问得到的一个临时寄存器，这一计算序列的执行也可以避免流水线停顿。

为了阐明上述栈访问对 Keccak-f1600 的性能影响，我们对两个版本的 Keccak-f1600 实现进行了测试。第一个版本不使用栈访问，但无法完全消除 RAW 数据冒险；第二个版本则如上文所述，使用栈访问来获取一个临时寄存器，从而能完全消除 RAW 数据冒险。我们使用三元组（时钟周期，指令数，CPI）来恒量程序的性能，第一个版本的指标为 $(2662, 5000, 0.53)$，第二个版本的指标为 $(2591, 5048, 0.51)$，这表明我们使用栈访问来获得一个临时寄存器的做法对 Keccak-f1600 的性能提升是有益的。第二个版本中，每一轮计算都需要使用一条 `ld` 和一条 `sd` 指令，因此相比于第一个版本多了 48 条指令。

RV64IB 上的双发射优化 RV64IB 上的双发射优化方法与 RV64I 上的类似，唯一的区别在于 RV64IB 上无需使用栈访问，因为 `rori` 指令的使用降低了寄存器分配压力。

指令统计 RV64I 上的 Keccak-f1600 实现每轮使用 202 条指令，包括 196 条逻辑指令、2 条 `ld/sd` 指令用于栈访问和 4 条指令用于读取 ι 步骤所需的常量以及维护相关内存地址。RV64IB 上的 Keccak-f1600 实现每轮使用 76 条 `xor` 指令、25 条 `andn` 指令和 29 条 `rori` 指令，共计 130 条逻辑指令和 4 条指令用于读取 ι 步骤所需的常量以及维护相关内存地址。

6.3.2 RV32I 和 RV32IB 上的优化实现

考虑到 RV32 仅提供了 $30 \times 32 = 960$ 比特的寄存器空间，无法将 1600 比特的 Keccak-f1600 状态常驻在寄存器中，因此 RV32 上的实现相比于 RV64 上的实现将更复杂。本节将以流水线友好性为第一目标来设计寄存器分配方案。RV32I 与 RV32IB 上的实现均采用 Becker 等人的就地实现方法；RV32I 实现利用了通道取补技术，RV32IB 则没有采用通道取补技术，具体原因与 RV64 上的实现类似；RV32I 没有使用位交错技术，而 RV32IB 则利用了该技术，具体原因见下

文。

```
.macro xor5 dst,b,g,k,m,s,tmp
  lw    \dst, \b(a0)
  lw    \tmp, \g(a0)
  xor   \dst, \dst, \tmp
  lw    \tmp, \k(a0)
  xor   \dst, \dst, \tmp
  lw    \tmp, \m(a0)
  xor   \dst, \dst, \tmp
  lw    \tmp, \s(a0)
  xor   \dst, \dst, \tmp
.endm
```

代码片段 6.3　Stoffelen 的 RV32I 上的部分实现代码

```
lw   tmp0l, 0*8+0(a0) # A[0,0]l
lw   tmp0h, 0*8+4(a0) # A[0,0]h
xor  tmp1l, A[0,1]l, A[0,2]l
xor  tmp1h, A[0,1]h, A[0,2]h
lw   tmp2l, 15*8+0(a0) # A[0,3]l
lw   tmp2h, 15*8+4(a0) # A[0,3]h
xor  tmp1l, \tmp1l, \tmp0l
xor  tmp1h, \tmp1h, \tmp0h
lw   tmp0l, 20*8+0(a0) # A[0,4]l
lw   tmp0h, 20*8+4(a0) # A[0,4]h
//...
```

代码片段 6.4　本章 RV32I 上的部分实现代码

Stoffelen 给出了 RV32I 上的一种 Keccak-f1600 实现[146]，其利用了文献[144] 中的 plane-by-plane 处理技术以及 4 轮展开就地实现方法，从而尽可能地减少内存访问。本节的实现没有采用类似的实现技术，具体原因为：

- 我们的目标之一是复用 RV32 上的实现来构造标量-向量混合实现，这需要我们的标量实现和向量实现具有类似的执行流程，因此促使我们使用 Becker 等人的就地实现方法。
- 我们的实现更注重流水线效率，Stoffelen 的实现存在大量的加载使用冒险（load-use hazards），其 CPI 为 0.69，距离理想最优值 0.5 有较大的差距。
- 使用 4 轮展开就地实现的可执行文件大小接近 Becker 等人就地实现的 4 倍，较大的可执行文件在某些情况下可能会导致性能降级。

寄存器分配方案和双发射优化　本节所使用的就地实现方法与 RV64 上的类似，下文中，我们使用 `loc` X 表示 X 所占用的寄存器或内存位置。由于 RV32 仅提供了 960 比特的寄存器空间，无法完全容纳 1600 比特的 Keccak-f1600 状态，所以频繁的内存访问是无法避免的，这又会增加潜在的加载使用数据冒险从而导致性能降级。

本节的基本策略是使部分状态常驻在寄存器中，从而使这些状态可直接用于计算，进而能和内存访问指令交替执行从而缓解加载使用数据冒险。在 RV64 实现中，每一轮的计算无需访问内存来获取 Keccak-f1600 状态，因此 a0 寄存器也被用于轮计算。然而，对于 RV32 实现，每一轮的计算都需要频繁地访问内存来读取或写入 Keccak-f1600 状态，因此 a0 寄存器无法被用于轮计算。

下文首先决定使用多少个寄存器来保存临时值。代码片段 6.4 基本阐明了本节的流水线优化策略，这一代码片段的执行不存在流水线停顿，共使用了 6 个临时寄存器。考虑到 RV32I 上实现 64 位旋转操作需要使用 2 个临时寄存器，因此用于保存临时值的寄存器数量至少为 8。除此以外，我们再分配 1 个寄存器用于标量-向量混合实现。

目前还剩余 20 个寄存器用于保存 640 比特 Keccak-f1600 状态，所选择的状态包括 $A[0,1]$, $A[0,2]$, $A[1,3]$, $A[1,4]$, $A[2,0]$, $A[2,3]$, $A[3,1]$, $A[3,4]$, $A[4,0]$, $A[4,2]$。在算法 6.1 的第二行中，每一个 $C[x]$ 的计算依赖于 5 个不同的状态值，即 $A[x,*]$。我们选择这些状态是为了保证计算每个 $C[x]$ 时都能直接从寄存器中得到 2 个 64 位状态，从而能充分地将内存访问指令与计算指令相交替来缓解加载使用数据冒险。代码片段 6.3 给出了 Stoffelen 的实现部分代码，该代码片段存在加载使用数据冒险；代码片段 6.4 给出了本节优化实现的部分代码片段，通过将内存访问指令与计算指令相交替来消除了加载使用数据冒险。

RV32I 上的 64 位旋转操作和位交错技术 我们使用 4 条移位指令和 2 条 `xor` 指令来实现 64 比特旋转操作，当偏移量 $n < 32$ 时，所需的指令序列为 `slli rl,al,n; srli al,al,32-n; srli rh,ah,32-n; slli ah,ah,n; xor rl,rl,rh; xor rh,ah,al`；当偏移量 $n \geq 32$ 时，所需的指令序列为 `slli rl,ah,n-32; srli ah,ah,64-n; srli rh,al,64-n; slli al,al,n-32; xor rl,rl,rh; xor rh,al,ah`。上述指令序列均不存在 RAW 数据冒险，不会导致流水线停顿。

下文分析位交错技术是否能带来性能改进。一个 32 位旋转操作消耗 2 条移位指令和 1 条 `xor` 指令。正如本章 6.2 所述，29 个 64 位旋转操作中的 6 个偏移量为 1，使用位交错技术能减少 $24 \times 6 \times 3 = 432$ 条指令。ML-KEM 和 ML-DSA 均使用 SHAKE128 来生成多项式矩阵，即使忽略 `absorbing` 阶段的编码开销，一次 `squeeze` 操作所需的解码操作消耗 1008 条指令。因此，位交错技术并不能给 RV32I 上的 Keccak-f1600 实现带来性能改进。

RV32IB 上的位交错技术 一个 64 位旋转操作依旧需要使用 4 条移位指令和 2 条 `xor` 指令。使用位交错技术后，29 个 64 位旋转操作中的 6 个偏移量为 1，故每个仅需使用 1 条 `rori` 指令；对于其余的 23 个 64 位旋转操作，每个需使用 2 条 `rori` 指令。因此节省的指令数为 $24 \times (29 \times 6 - 6 - 23 \times 2) = 2928$。因此，位交错技术对于 RV32IB 上的 Keccak-f1600 实现是有益的，并且还能减轻寄存器分配压力。

指令统计 RV32I 实现的每一轮消耗 344 条逻辑指令、115 条 `lw` 指令和 101 条 `sw` 指令。RV32IB 实现的每一轮消耗 258 条逻辑指令、115 条 `lw` 指令和 101 条 `sw` 指令。大部分内存访问指令用于读取和写入 Keccak-f1600 状态，少部分用于栈访问。

6.3.3 RVV 上的优化实现

使用向量指令实现 Keccak-f1600 的基本策略与 RV64I 上的实现策略类似，RV64I 上的标量逻辑指令均有对应的向量版本。考虑到本章所使用的玄铁 C908 的 `VLEN=128`，因此向量实现版本一次可并行执行两路 Keccak-f1600 。

RVV 上的实现与 RV64I 上的实现的区别在于：(1) RVV 指令集具有 32 个向量寄存器，拥有足够的寄存器进行流水线优化，并且无需栈访问。(2) 向量移位指令的 `vi` 版本，比如 `vsll.vi vd,vs2,uimm`，其中的 `uimm` 是 5 比特的立即数，用于指定移位偏移量。考虑到 Keccak-f1600 实现过程中所需的移位偏移量可能会超出 5 比特，因此我们不得不使用移位指令的 `vx` 版本，即使用一个标量寄存器来指定移位偏移量。因此，RVV 实现中需使用少量的 `li` 指令来构造移位偏移量，并且需要占用一个标量寄存器。

指令统计 RVV 实现的 Keccak-f1600 每一轮使用 196 条向量逻辑指令、1 条 `vle64` 指令用于读取 ι 步骤所需的常量和 26 条标量指令。

6.3.4 RV{32,64}I{B}{V} 上的标量-向量混合实现

标量-向量混合实现技术曾被用于加速 ARMv7-A 架构上的 Salsa20 算法[22]、ARMv8-A 架构上的 X25519 算法[147] 以及 ARMv8-A 和 ARMv8.4-A 架构上的 Keccak-f1600 算法[145]。正如 Becker 等人所述[145]：标量-向量混合实现涉及到交替执行代码路径 A 和 B，他们使用不同的处理器执行单元，因此可以并行执行。混合实现能改进程序执行效率的原因是：这些处理器往往具有独立的标量执行单元和 SIMD 执行单元，本章所使用的 C908 处理器也具有这一特性。

Becker 等人从 IPC（instructions per cycle）角度对混合实现的解释为：程序 A 和 B 的 IPC 分别为 w_A 和 w_B，w_{max} 为目标平台的最大 IPC，当 $w_A + w_B < w_{max}$ 时混合实现能带来性能提升。

混合实现的可行性以及构造方法依赖于处理器的微架构，下文均以玄铁 C908 处理器为例展开研究，其微架构细节以及指令时延在本文 5.2.1 中给出了讨论。下文首先分析 RVV 上 Keccak-f1600 实现的性能，然后分别论证混合实现在 RV64I{B}V 和 RV32I{B}V 上的可行性。

RVV 上的实现性能分析 正如本文 5.2.1 所述，C908 处理器的向量指令只支持单发射，并且向量逻辑指令的 CPI 为 2，因此向量实现的 Keccak-f1600 的理想最优 CPI 为 2。表 6.2 给出了本章 RISC-V 上 Keccak-f1600 实现的性能数据，从中可知 RVV 上 Keccak-f1600 的 CPI 为 1.77，略低于 2 的原因是该实现中使用了一些标量指令。对于 RVV 实现，单路 Keccak-f1600 的平均开销为 4827 时钟周期，明显慢于 RV64I 和 RV64IB 实现，主要原因是 C908 处理器的向量逻辑计算能力较弱。

表 6.1　C908 处理器 RV32I{B} 和 RV64I{B} 上 Keccak-f1600 的时钟周期对比

实现	技术组合	时钟周期	指令数	CPI
RV32I 上 C 实现	参考实现	15779	24487	0.64
RV32I 上 Stoffelen 的实现[146]	通道取补 & 位交错 & 4 轮就地实现	8734	12740	0.69
RV32I 上本章实现	通道取补 & 单轮就地实现 & 双发射优化	7808	14890	0.52
RV32IB 上 C 实现	参考实现	12341	20460	0.6
RV32IB 上本章实现	位交错 & 单轮就地实现 & 双发射优化	6222	11554	0.54
RV64I 上 C 实现	参考实现	4926	8585	0.57
RV64I 上本章实现	通道取补 & 单轮就地实现 & 双发射优化	2591	5049	0.51
RV64IB 上 C 实现	参考实现	2412	4296	0.56
RV64IB 上 RISCV-Crypto 实现[1]	内敛汇编 `rori/andn`	2563	4649	0.55
RV64IB 上本章实现	单轮就地实现 & 双发射优化	1770	3405	0.52

[1] https://github.com/riscv/riscv-crypto/blob/main/benchmarks/sha3/zscrypto_rv64/Keccak.c；commit efb77a9

RV64I{B}V 上的混合实现　正如表 6.1 所述，本章 RV64I 和 RV64IB 实现的 CPI 分别为 0.51 和 0.52，接近理想最优值 0.5，意味着 RV64I 和 RV64IB 实现均充分利用了 CPU 的前端和后端。如果交替执行 RVV 实现和 RV64I{B} 实现来构造混合实现，向量指令会和标量指令竞争 CPU 的前端能力。尽管一条向量指令能并行执行两路 64 位操作，但 C908 处理器的向量逻辑计算能力弱于标量指令，因此混合实现方法对于 RV64I{B}V 上的实现无法产生性能改进。

RV32I{B}V 上的混合实现　C908 处理器的最优 CPI 为 $w_{max} = 0.5$，本章 RV32I 实现的 CPI 为 $w_A = 0.52$，RVV 实现的 CPI 为 $w_B = 1.77$。混合实现对本章 RV32I{B}V 上的实现有性能改进的一个前提是：RVV 实现计算单路 Keccak-f1600 要快于 RV32I{B} 实现。考虑到 C908 处理器上向量逻辑指令的计算时延为 4 个时钟周期，CPI 为 2，因此可理解为具有 2 个独立的向量逻辑执行单元。考虑到连续执行 3 条向量逻辑指令，第三条指令将不得不停顿 2 个时钟周期来等待可用的执行单元。因此对于 RVV 实现的 Keccak-f1600，即使处理器前端每个时钟周期能发射

表 6.2　C908 处理器 RVV 上 Keccak-f1600 的时钟周期对比，括号中的数据根据并行度进行了标准化处理

实现	技术组合	时钟周期	指令数	CPI
本章 RVVx2 实现	通道取补 & 单轮就地实现 & 双发射优化	9655 (4827)	5462	1.77
Becker 等人的 A55-NEON 实现[145]	单轮就地实现	4560 (2280)	3840	1.19
Becker 等人的混合 x5 实现[145]	A55-Scalar & A55-NEON	8960 (1792)	-	-
本章 RV32IVx3 实现	RV32I & RVVx2	11850 (3950)	20273	0.58
本章 RV32IVx4 实现		20374 (5093)	35190	0.58
本章 RV32IVx5 实现		30544 (6108)	50285	0.61
本章 RV32IBVx3 实现	RV32IB & RVVx2	10527 (3509)	17012	0.62
本章 RV32IBVx4 实现		16670 (4167)	28472	0.59
本章 RV32IBVx5 实现		24299 (4859)	39991	0.61

一条指令，但后端的逻辑执行单元是性能瓶颈，会导致流水线停顿影响程序性能。我们对指令序列 `vand;vand` 和 `vand;vand;and;and;and;and` 进行了测试，这两个指令序列的耗时均为 4 个时钟周期，佐证了上述说法。

在理想情况下，向量指令数与标量指令数的比例为 1 : 2 可以实现最佳性能。本章 RVV 实现每轮使用 197 个向量指令和 26 个标量指令；而 RV32I 和 RV32IB 每轮分别使用 560 和 474 条标量指令。使用 1 个 RVV 实现和 1 个 RV32I/RV32IB 来构造混合实现时，向量指令数与标量指令数的比例分别为 1 : 3 和 1 : 2.5。使用 3 个 RVV 实现和 2 个 RV32I 实现能构造出理想的指令比例，但其一次计算 8 路并行的 Keccak-f1600，既增加了编程复杂性还很难构造统一的逻辑执行流。因此，我们决定使用 1 个 RVV 实现和 1/2/3 个 RV32I/RV32IB 实现来构造混合实现，最终实现了 3/4/5 路的 Keccak-f1600 混合实现。由表 6.2 可知，3 路 Keccak-f1600 实现的性能最优。

6.4　性能评估

用于性能评估的处理器是支持各种 RISC-V 指令集组合的玄铁 C908 处理器，在本文 5.2.1 中给出了介绍，因此不再赘述。如无特殊说明，下文中的时钟周期数据均是通过多次运行对应程序取均值或中间值得到。

本节 6.4.1 给出了 Keccak-f1600 在各种处理器上的性能对比。本节 6.4.2 和 6.4.3 分别给出了 ML-KEM 和 ML-DSA 的性能数据以及对比，为了方便与相关工作进行对比，本文的 ML-KEM 与 ML-DSA 实现分别基于 Kyber[1]和 Dilithium[2]的代码进行开发，并且集成了本文第五章基于改进版 Plantard 模乘的 NTT 实现与本章优化的 Keccak-f1600 实现。

Keccak-f1600 向 ML-KEM 和 ML-DSA 的集成 标量实现的 Keccak-f1600 均为单路实现，因此集成较为直接。而标量-向量混合实现是多路的，以集成 RV32IVx3 实现为例：ML-KEM-512 `KeyGen` 需要生成维度为 2×2 的多项式矩阵，即生成 4 个多项式，因此使用 1 次 RV32IVx3 实现和 1 次 RV32I 标量实现；其余的集成思路与此类似。

6.4.1 Keccak-f1600 性能对比

玄铁 C908 处理器 RV32I{B} 和 RV64I{B} 表 6.1 给出了 C908 处理器 RV32I{B} 和 RV64I{B} 上 Keccak-f1600 实现的性能数据，其中参考 C 实现从 PQClean 开源代码库[3]获取。

Stoffelen 在 RV32I 上的实现[146] 基于 XKCP 开源库[4]，所使用的技术包括位交错、通道取补和 4 轮就地实现，并且还利用了逐平面（plane-by-plane）处理技术来减少栈访问。需要注意的是，ARMv7-M 支持旋转指令，因此位交错技术能带来性能改进，而在 RV32I 上位交错技术则不能带来直接的性能改进。

本章 RV32I 优化实现的速度是参考 C 实现的 2 倍，并且比 Stoffelen 的实现快 12%。需要注意的是，Stoffelen 的实现利用了位交错技术，以 SHAKE128 为例，该技术的使用意味着 `squeeze` 操作需额外使用至少 1008 条指令来处理编解码。

得益于 B 扩展提供的 `rori` 和 `andn` 指令，本章 RV32IB 实现比参考 C 实现快 98%，比本章 RV32I 实现快 25%。

本章 RV64I 优化实现比参考 C 实现快 90%。参考 C 实现在 RV64IB 指令集上也有不俗的性能表现，这是因为编译器也充分地利用了 B 扩展所提供的 `rori` 和 `andn` 指令，这两条指令降低了寄存器分配压力，因此减少了栈访问。本章优化的 RV64IB 实现比参考 C 实现快 36%。

玄铁 C908 处理器 RVV 表 6.2 给出了 C908 处理器 RVV 上 Keccak-f1600 实现以及混合实现的性能数据。正如本文 5.2.1 所述，C908 处理器计算向量逻辑指令的能力较弱，因此 RVV 实现的 CPI 较高。本章混合实现能带来性能改进是得益于标量指令与向量指令的交替执行缓解了 RVV

[1]https://github.com/pq-crystals/kyber/tree/main/ref; commit 441c051.

[2]https://github.com/pq-crystals/dilithium/tree/master/ref; commit f1f8085.

[3]https://github.com/PQClean/PQClean/blob/master/common/fips202.c；commit 05df469。

[4]https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/ARM/KeccakP-1600-inplace-32bi-armv7m-le-gcc.s

表 6.3　C908 处理器上 ML-KEM-768 的时钟周期（k=1000）对比，KeyGen、Encaps 与 Decaps 分别表示密钥生成、封装与解封装

实现	KeyGen	Encaps	Decaps
Huang 等人在 RV32IM 上的实现[132]	1052k	1261k	1179k
RV32IM 上的参考 C 实现[1]	1222k	1602k	1691k
RV32IMB 上的参考 C 实现[1]	1048k	1377k	1481k
本章 RV32IM 上的优化实现	504k	614k	589k
本章 RV32IMB 上的优化实现	445k	554k	537k
本章 RV32IMV 上的优化实现	316k	422k	375k
本章 RV32IMBV 上的优化实现	290k	387k	352k
RV64IM 上的参考 C 实现[1]	742k	986k	1185k
RV64IMB 上的参考 C 实现[1]	603k	803k	1011k
本章 RV64IM 上的优化实现	279k	328k	358k
本章 RV64IMB 上的优化实现	237k	275k	316k
本章 RV64IMV 上的优化实现	209k	249k	253k
本章 RV64IMBV 上的优化实现	169k	201k	213k

[1] https://github.com/pq-crystals/kyber/tree/main/ref；commit 441c051.

实现的流水线停顿。本章的 RV32IVx3 混合实现性能比 RVV 实现快 22%，比本章 RV32I 实现快 98%，是 RV32I 参考 C 实现的 4 倍。本章的 RV32IBVx3 混合实现性能比 RVV 实现快 38%，比本章 RV32IB 实现快 77%，是 RV32IB 参考 C 实现的 3.5 倍。

6.4.2　ML-KEM 方案性能对比

玄铁 C908 处理器　表 6.3 给出了 C908 处理器上 ML-KEM-768 的时钟周期对比：

- RV32IM：本工作的性能是 Huang 等人 RV32IM 实现[132] 的 2 倍 ~ 2.1 倍，是参考实现的 2.4 倍 ~ 2.9 倍。
- RV32IMB：本工作的性能是参考实现的 2.4 倍 ~ 2.8 倍。
- RV32IMV：本工作 RV32IMV 实现性能是本工作 RV32IM 实现的 1.5 倍 ~ 1.6 倍；是 RV32IM 参考实现的 3.8 倍 ~ 4.5 倍。
- RV32IMBV：本工作 RV32IMBV 实现性能是本工作 RV32IMB 实现的 1.4 倍 ~ 1.5 倍；是 RV32IMB 参考实现的 3.6 倍 ~ 4.2 倍。
- RV64IM：本工作的性能是参考实现的 2.7 倍 ~ 3.3 倍。
- RV64IMB：本工作的性能是参考实现的 2.5 倍 ~ 3.2 倍。

表 6.4　C908 处理器上 ML-DSA-65 的时钟周期对比，KeyGen、Sign 与 Verify 分别表示密钥生成、签名与验证签名

实现	KeyGen	Encaps	Decaps
RV32IM 上的参考实现[1]	4123k	13671k	4145k
RV32IMB 上的参考实现[1]	3422k	12635k	3504k
本章 RV32IM 上的优化实现	1938k	5050k	1889k
本章 RV32IMB 上的优化实现	1752k	4656k	1721k
本章 RV32IMV 上的优化实现	1222k	3281k	1207k
本章 RV32IMBV 上的优化实现	1149k	3120k	1134k
RV64IM 上的参考实现[1]	1841k	8232k	1958k
RV64IMB 上的参考实现[1]	1328k	7217k	1474k
本章 RV64IM 上的优化实现	916k	3326k	941k
本章 RV64IMB 上的优化实现	752k	3067k	790k
本章 RV64IMV 上的优化实现	879k	2474k	850k
本章 RV64IMBV 上的优化实现	718k	2189k	694k

[1] https://github.com/pq-crystals/dilithium/tree/master/ref；commit f1f8085.

- RV64IMV：本工作 RV64IMV 实现性能是本工作 RV64IM 实现的 1.3 倍 ~ 1.4 倍；是 RV64IM 参考实现的 3.6 倍 ~ 4.7 倍。
- RV64IMBV：本工作 RV64IMBV 实现性能是本工作 RV64IMB 实现的 1.4 倍 ~ 1.5 倍；是 RV64IMB 参考实现的 3.6 倍 ~ 4.7 倍。

6.4.3　ML-DSA 方案性能对比

玄铁 C908 处理器　表 6.4 分别给出了 C908 处理器上 ML-DSA-65 的时钟周期对比：

- RV32IM：本工作的性能是参考实现的 2.1 倍 ~ 2.7 倍。
- RV32IMB：本工作的性能是参考实现的 2.0 倍 ~ 2.7 倍。
- RV32IMV：本工作 RV32IMV 实现性能是本工作 RV32IM 实现的 1.5 倍 ~ 1.6 倍；是 RV32IM 参考实现的 3.4 倍 ~ 4.2 倍。
- RV32IMBV：本工作 RV32IMBV 实现性能是本工作 RV32IMB 实现的 1.5 倍；是 RV32IMB 参考实现的 3.0 倍 ~ 4.0 倍。
- RV64IM：本工作的性能是参考实现的 2.1 倍 ~ 2.5 倍。
- RV64IMB：本工作的性能是参考实现的 1.8 倍 ~ 2.4 倍。

- RV64IMV：本工作 RV64IMV 实现性能是本工作 RV64IM 实现的 1.04 倍 ~ 1.34 倍；是 RV64IM 参考实现的 2.1 倍 ~ 3.3 倍。
- RV64IMBV：本工作 RV64IMBV 实现性能是本工作 RV64IMB 实现的 1.05 倍 ~ 1.4 倍；是 RV64IMB 参考实现的 1.8 倍 ~ 3.3 倍。

6.5 本章小结

SHA-3 是格密码 ML-KEM 与 ML-DSA 方案的性能瓶颈之一；RISC-V 架构的模块化设计原则也带来了指令集组合碎片化的问题，SHA-3 在各种不同指令集组合上的优化实现研究仍不够充分。为了解决上述问题，本章研究了 SHA-3 算法的核心运算 Keccak-f1600 在 8 种不同的 RISC-V 指令集组合上的性能优化实现。本章详细地回顾了用于优化实现 Keccak-f1600 的各种技术，并为不同嵌入式处理器上的 Keccak-f1600 实现设计了最优的技术组合。实验结果表明，本章的研究刷新了 Keccak-f1600 在各种嵌入式处理器上的性能记录，性能提升最高可达之前实现的 4 倍。此外，我们将本章优化的 Keccak-f1600 实现与第五章优化的 NTT 实现集成至 NIST 后量子密码标准 ML-KEM 和 ML-DSA 方案中，结果表明，ML-KEM 的性能最高是之前实现的 4.7 倍，ML-DSA 的性能最高是之前实现的 4.2 倍。考虑到 Keccak-f1600 在各种 RISC-V 指令集上的优化实现研究较少，本章的研究填补了这一空白，丰富了 RISC-V 处理器上密码实现的生态。除此以外，本章 Keccak-f1600 的优化实现不仅能提升 ML-KEM 和 ML-DSA 等格密码的性能，还能使基于哈希的数字签名方案（如 XMSS、LMS 和 SPHINCS$^+$）受益。综上所述，该研究对于后量子密码在各种嵌入式处理器上的应用和普及具有重要推动作用。

第七章　全文总结与展望

7.1　全文总结

本文主要研究椭圆曲线密码（ECC）和后量子密码（PQC）这两类公钥密码算法的优化实现。ECC 相较 RSA 具有安全性高、计算复杂度低和密钥长度短等优势，已在诸多网络协议中广泛应用；后量子密码则为应对量子计算威胁下的网络空间安全提供有效方案，二者结合有望持续保障未来网络空间的数据安全。

针对 ECC 的研究，本工作涉及我国商密标准 SM2 及国际标准 X/Ed25519，旨在解决的问题和面临的挑战包括：（1）SM2 在 x86-64 架构上的优化实现已得到充分的关注和研究，但在国产 ARM 处理器上性能欠佳，仍有较大的改进空间；（2）ECC 的核心运算不具备直观的并行计算流程，难以充分利用诸如 Intel AVX-512 这类 SIMD 指令集的并行计算能力；（3）密码工程领域的大部分研究通常仅关注密码算法的性能优化，而往往忽略了研究如何将优化实现集成到复杂的 TLS 软件栈中；（4）Intel AVX2/AVX-512 虽然提供了强大的并行计算能力，但其硬件单元存在冷启动问题，在 TLS 握手场景中导致密码算法遭受了严重的性能降级。

针对 PQC 的研究，本工作主要针对基于模格的 Saber 方案和 NIST 所制定的 ML-KEM 与 ML-DSA 标准，研究动机包括：（1）Saber 方案原本的参数设定不支持高效的 NTT 多项式乘法，如何为其适配 NTT 算法并改进该方案的性能？（2）基于模格的密码方案涉及到占用较大内存资源的多项式矩阵，这导致其难以部署在内存资源严重受限的物联网设备上，如何对其进行内存优化从而满足物联网场景下的部署条件？（3）对于 NTT 多项式乘法的性能优化，目前大部分研究主要集中在单发射处理器上的优化，如何在双发射 RISC-V 处理器上对 NTT 多项式乘法进行性能优化？（4）SHA-3 的核心组件 Keccak-f1600 计算复杂且寄存器资源需求高，如何在 RISC-V 处理器上对其进行高效实现？

为了解决上述问题，本工作从四个方面对 ECC 和 PQC 的优化实现展开研究：

第一，本文针对 SM2 在华为鲲鹏 ARM 处理器上进行了深入的分析和优化。在有限域层面，本工作基于 SM2 模数的数值特征推导了优化的蒙哥马利模乘 CIOS 方法，并设计了更快的基于费马小定理的模逆算法；在标量乘层面，本工作分别为固定点与非固定点标量乘实现了宽度为 7 和 5 的窗口算法，固定点标量乘得益于预计算技术的使用，性能是之前实现的 30 多倍；对于签名生成的计算，本工作提出了更快的计算方法，可使用一个模加/减运算替换模乘运算。最终将上述优化技术集成到 OpenSSL 中后的测试表明，在华为云鲲鹏 920 处理器上，SM2 签名与验签性能分别是之前实现的 8.7 倍和 3.5 倍。该工作可在一定程度上推进 SM2 在国产 ARM 服务器上的应用和普及。

第二，本文针对 X/Ed25519 在 Intel AVX-512 指令集上进行了自底向上的并行优化，并研究了如何将优化实现集成到复杂的 TLS 软件栈中以及如何缓解冷启动问题。对于有限域运算，本工作设计并实现了 8 路并行策略以最大化 AVX-512 指令集的并行计算能力，并利用形式化验证工具 CryptoLine 对其进行验证以确保正确性和鲁棒性。对于椭圆曲线点运算和标量乘运算，本工作充分探讨了各种并行策略的可行性。得益于自底向上的并行优化策略，本工作的 X/Ed25519 实现性能最多是 OpenSSL 实现性能的 12 倍。除此以外，本工作还基于 OpenSSL ENGINE API 设计实现了 ENG25519 引擎，从而在不修改 TLS 软件栈以及应用代码的前提下将优化的 X/Ed25519 实现集成到 TLS 应用中，使 TLS 应用能受益于本工作的性能改进。为了缓解冷启动问题，本工作开发了一个基于启发式 “热身” 方案的辅助线程。最终，端到端性能测试表明，DNS over TLS 服务端的峰值吞吐至多可提升 41%。该工作能使大规模互联网服务提供商在提升用户体验、增强服务可用性以及降低运营成本等方面受益。

第三，本文研究了如何为 Saber 方案适配 NTT 算法，以及如何在双发射 RV32IM、双发射 RV64IM 和 RVV 处理器上优化 ML-KEM 与 ML-DSA 方案的 NTT 多项式乘法。Saber 方案原本的参数设定并不支持 NTT 多项式乘法，本工作研究了如何为 Saber 适配 NTT 算法并研究了 Saber 方案的内存优化方法，本章的实现不仅刷新了性能记录，还减少了 Saber 的内存占用，有助于推动 Saber 方案在内存受限的物联网设备上的部署与应用。相比于广泛使用的蒙哥马利和 Barrett 算法，改进版 Plantard 模乘算法在计算效率、输入输出范围等方面具有优势，但其严格依赖于 $l \times 2l$ 乘法器。本工作指出，ML-KEM 方案的 NTT 在 RV32IM 和 RV64IM 上可使用改进版 Plantard 模乘，ML-DSA 方案的 NTT 只能在 RV64IM 上使用改进版 Plantard 模乘，除此以外均推荐使用蒙哥马利模乘进行实现。本工作重点研究了 NTT 多项式乘法在双发射处理器上的流水线优化方法，提出的流水线调优技术能减少因乘法指令耗时较高所导致的流水线停顿。最终的性能测试表明，本章的实现均刷新了性能记录，以 ML-KEM 方案的 NTT 多项式乘法在玄铁 C908 处理器上的实现为例，本章的优化实现性能最高可达之前实现的 29.9 倍。该工作能有效促进格密码在各种嵌入式处理器上的应用和普及，并对我国国产自主可控的后量子密码在嵌入式平台上的高效实现与实际部署具有一定的启发作用。

第四，本工作研究了格密码 ML-KEM 与 ML-DSA 方案中的另一耗时运算 SHA-3 的性能优化。SHA-3 的核心组件是 Keccak-f1600 置换算法，本章在双发射 RISC-V 处理器上的 Keccak-f1600 优化实现涵盖了各种常见的指令集组合，并重新审视了各种优化实现技术，从而为各种指令集构造最优技术组合。对于 RV64I 和 RV64IB 上的实现，一方面本工作阐述了如何进行合理的流水线调度以减少流水线阻塞；另一方面本工作阐明了如何利用 B 扩展提供的 `rori` 和 `andn` 指令来加速 Keccak-f1600 。对于 RV32I 和 RV32IB 上的实现，本工作以流水线友好性为主要目标，设计了专门的寄存器分配方案和双发射流水线优化方法，从而解决寄存器资源不足的难题。本工作还探讨了标量-向量混合实现技术在各种指令集组合上的可行性。实验测试表明，本工作

的实现相比于之前已知的最快实现性能提升至多可达 98%。最终将优化的 NTT 实现与优化的 Keccak-f1600 实现集成到 ML-KEM 与 ML-DSA 方案中后，性能分别至高可达之前实现的 4.7 倍和 4.2 倍。该工作有望进一步丰富 SHA-3 以及后量子密码在 RISC-V 上的软件生态。

7.2 未来工作展望

在多项式乘法优化实现方面，本工作主要研究 NTT 算法在双发射 RISC-V 处理器上的性能优化，玄铁 C908 处理器的向量单元的硬件配置为 VLEN=128，即每个向量寄存器的长度为 128 比特，NTT 多项式乘法在 VLEN=256 的处理器上的优化实现有待进一步研究。

在 Keccak-f1600 优化实现方面，尤其是 RV32I 与 RV32IB 上的实现，本工作发现使用自动化方法来进行流水线调优是可行的，比如针对 ARM 处理器设计的 SLOTHY 工具即可在一定程度上对流水线调优进行自动化，因此以自动化的形式探索 Keccak-f1600 的流水线调优值得深入研究。

本工作也能对我国未来自主可控的后量子密码标准起到一定的启发作用。本工作的性能优化思路可以应用于我国自主设计的后量子密码方案中，诸如 AKCN-MLWE、CTRU 和 LAC 等密码方案。

本工作所研究的 Intel AVX2/AVX-512 硬件单元的冷启动问题值得进一步开展研究，如下问题值得进一步探索：（1）AMD 处理器上的 AVX2/AVX-512 硬件单元是否存在该问题？（2）ARMv8-A 架构上的 NEON 硬件单元是否存在该问题？（3）考虑到冷启动问题的影响，在某些场景下是否存在标量实现性能优于冷启动情况下的向量实现的可能？

参考文献

[1] Koblitz N. Elliptic curve cryptosystems[J]. Mathematics of computation, 1987, 48(177):203–209.

[2] Miller V S. Use of elliptic curves in cryptography[C]. Proceedings of Conference on the theory and application of cryptographic techniques. Springer, 1985. 417–426.

[3] Rescorla E, Dierks T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August, 2008. https://www.rfc-editor.org/info/rfc5246.

[4] Rescorla E. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. https://www.rfc-editor.org/info/rfc8446, August, 2018.

[5] Lonvick C M, Ylonen T. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January, 2006. https://www.rfc-editor.org/info/rfc4253.

[6] Bider D. Use of RSA Keys with SHA-256 and SHA-512 in the Secure Shell (SSH) Protocol. RFC 8332, March, 2018. https://www.rfc-editor.org/info/rfc8332.

[7] Harris B, Velvindron L. Ed25519 and Ed448 Public Key Algorithms for the Secure Shell (SSH) Protocol. RFC 8709, February, 2020. https://www.rfc-editor.org/info/rfc8709.

[8] Baushke M D. Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH). RFC 9142, January, 2022. https://www.rfc-editor.org/info/rfc9142.

[9] Frankel S, Krishnan S. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071, February, 2011. https://www.rfc-editor.org/info/rfc6071.

[10] Zhong H S, Wang H, Deng Y H, et al. Quantum computational advantage using photons[J]. Science, 2020, 370(6523):1460–1463.

[11] NIST. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final, 2024.

[12] NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/204/final, 2024.

[13] NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard. https://csrc.nist.gov/pubs/fips/205/final, 2024.

[14] Avanzi R, Bos J, Ducas L, et al. CRYSTALS-Kyber (version 3.02) –Submission to round 3 of the NIST post-quantum project. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf, 2021.

[15] Ducas L, Kiltz E, Lepoint T, et al. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018, 2018(1):238–268.

[16] Aumasson J P, Bernstein D J, Beullens W, et al. SPHINCS+ (version 3.1) - Submission to round 3 of the NIST post-quantum project. https://sphincs.org/data/sphincs+-r3.1-specification.pdf, 2022.

[17] Gura N, Patel A, Wander A, et al. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs[C]. In: Joye M, Quisquater J, (eds.). Proceedings of Cryptographic Hardware and Embed-

ded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004. 119–132.

[18] NIST. Digital Signature Standard (DSS), FIPS 186-2. https://csrc.nist.gov/files/pubs/fips/186-2/final/docs/fips186-2.pdf, 2000.

[19] Langley A, Hamburg M, Turner S. RFC 7748: Elliptic Curves for Security. https://www.rfc-editor.org/info/rfc7748, January, 2016.

[20] Masny D, Rindal P. Endemic Oblivious Transfer[C]. Proceedings of Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019. ACM, 2019. 309–326.

[21] Koc C K, Acar T, Kaliski B S. Analyzing and comparing Montgomery multiplication algorithms[J]. IEEE micro, 1996, 16(3):26–33.

[22] Bernstein D J, Schwabe P. NEON Crypto[C]. Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2012, volume 7428. Springer, 2012. 320–339.

[23] Mai L, Yan Y, Jia S, et al. Accelerating SM2 Digital Signature Algorithm Using Modern Processor Features[J]. Proceedings of the International Conference on Information and Communications Security, 2020. 430–446.

[24] Gueron S, Krasnov V. Fast prime field elliptic-curve cryptography with 256-bit primes[J]. Journal of Cryptographic Engineering, 2015, 5(2):141–151.

[25] Faz-Hernández A, López J, Dahab R. High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions[J]. ACM Transactions on Mathematical Software,, 2019, 45(3):25:1–25:35.

[26] Cheng H, Großschädl J, Tian J, et al. High-Throughput Elliptic Curve Cryptography Using AVX2 Vector Instructions[C]. Proceedings of Selected Areas in Cryptography - SAC 2020, volume 12804. Springer, 2020. 698–719.

[27] OpenSSH. OpenSSH 9.0 was released on 2022-04-08. https://www.openssh.com/txt/release-9.0, 2022.

[28] MullVad. Quantum-Resistant Tunnels Now Available on iOS! https://mullvad.net/en/blog/quantum-resistant-tunnels-now-available-on-ios, 2024.

[29] Thipsay A. Microsoft's quantum-resistant cryptography is here. https://techcommunity.microsoft.com/t5/security-compliance-and-identity/microsoft-s-quantum-resistant-cryptography-is-here/ba-p/4238780, 2024.

[30] Seiler G. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography[J]. Cryptology ePrint Archive, 2018..

[31] Alkim E, Ducas L, Pöppelmann T, et al. Post-quantum Key Exchange - A New Hope[C]. Proceedings of 25th USENIX Security Symposium (USENIX Security 16), 2016. 327–343.

[32] Becker H, Hwang V, Kannwischer M J, et al. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, 2022(1):221–244.

[33] Botros L, Kannwischer M J, Schwabe P. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4[C]. Proceedings of Progress in Cryptology - AFRICACRYPT 2019, volume 11627. Springer, 2019. 209–228.

[34] Greconici D O C, Kannwischer M J, Sprenkels A. Compact Dilithium Implementations on

Cortex-M3 and Cortex-M4[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020, 2021(1):1–24.

[35] Alkim E, Bilgin Y A, Cenk M, et al. Cortex-M4 optimizations for {R, M} LWE schemes[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020, 2020(3):336–357.

[36] Greconici D. Kyber on RISC-V[D]. 2020.

[37] 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[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, 2021(2):159–188.

[38] Albrecht M R, Hanser C, Höller A, et al. Implementing RLWE-based Schemes Using an RSA Co-Processor[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019, 2019(1):169–208.

[39] Moody D, Perlner R, Regenscheid A, et al. Transition to Post-Quantum Cryptography Standards, 2024. https://doi.org/10.6028/NIST.IR.8547.ipd.

[40] 国家密码管理局. 国家密码管理局关于发布《SM2 椭圆曲线公钥密码算法》公告. https://oscca.gov.cn/sca/xxgk/2010-12/17/content_1002386.shtml, 2010. Accessed: 2024-04-08.

[41] ISO. ISO/IEC 14888-3:2018. https://www.iso.org/obp/ui/#iso:std:iso-iec:14888:-3:ed-4:v1:en, 2018. Accessed: 2024-04-08.

[42] Hankerson D, Menezes A, Vanstone S. Guide to elliptic curve cryptography. 第一版 [M]. Springer Science & Business Media, 2006.

[43] Montgomery P L. Speeding the Pollard and elliptic curve methods of factorization[J]. Mathematics of computation, 1987, 48(177):243–264.

[44] Bernstein D J. Curve25519: New Diffie-Hellman Speed Records[C]. Proceedings of Public Key Cryptography - PKC 2006, volume 3958. Springer, 2006. 207–228.

[45] Hisil H, Wong K K, Carter G, et al. Twisted Edwards Curves Revisited[C]. Proceedings of Advances in Cryptology - ASIACRYPT 2008, volume 5350. Springer, 2008. 326–343.

[46] Bernstein D J, Duif N, Lange T, et al. High-speed high-security signatures[J]. Journal of Cryptographic Engineering, 2012, 2(2):77–89.

[47] Finney H, Donnerhacke L, Callas J, et al. OpenPGP Message Format. RFC 4880, November, 2007. https://www.rfc-editor.org/info/rfc4880.

[48] Dempsky M. DNSCurve: Link-Level Security for the Domain Name System[R]. Internet-Draft draft-dempsky-dnscurve-01, Internet Engineering Task Force, February, 2010. https://datatracker.ietf.org/doc/draft-dempsky-dnscurve/01/. Work in Progress.

[49] Marlinspike M, Perrin T. The X3DH Key Agreement Protocol. https://www.signal.org/docs/specifications/x3dh/, 2016. Accessed: 2024-04-08.

[50] Marlinspike M, Perrin T. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016. Accessed: 2024-04-08.

[51] Barnes R, Beurdouche B, Robert R, et al. The Messaging Layer Security (MLS) Protocol. RFC 9420, July, 2023. https://www.rfc-editor.org/info/rfc9420.

[52] Bernstein D J. 25519 naming. https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/, 2014. Accessed: 2023-07-01.

[53] Josefsson S, Liusvaara I. RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA). https://www.rfc-editor.org/info/rfc8032, January, 2017.

[54] Fujisaki E, Okamoto T. Secure Integration of Asymmetric and Symmetric Encryption Schemes[J]. J. Cryptol., 2013, 26(1):80–101.

[55] Hofheinz D, Hövelmanns K, Kiltz E. A Modular Analysis of the Fujisaki-Okamoto Transformation[C]. In: Kalai Y, Reyzin L, (eds.). Proceedings of Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I, volume 10677 of *Lecture Notes in Computer Science*. Springer, 2017. 341–371.

[56] Basso A, Mera J M B, D' Anvers J, et al. Saber: Mod-LWR based KEM (Round 3 Submission), 2020.

[57] D'Anvers J, Karmakar A, Roy S S, et al. Saber: Module-LWR Based Key Exchange, CPA-secure Encryption and CCA-secure KEM[C]. Proceedings of Progress in Cryptology - AFRICACRYPT 2018. Springer, 2018. 282–305.

[58] Banerjee A, Peikert C, Rosen A. Pseudorandom Functions and Lattices[C]. Proceedings of Advances in Cryptology - EUROCRYPT 2012, volume 7237. Springer, 2012. 719–737.

[59] Varma C. A study of the ECC, RSA and the Diffie-Hellman algorithms in network security[C]. Proceedings of 2018 International Conference on Current Trends towards Converging Technologies (ICCTCT). IEEE, 2018. 1–4.

[60] OpenSSL-Project. OpenSSL library. https://www.openssl.org/, 2024. Accessed: 2024-04-08.

[61] GmSSL-Project. GmSSL library. https://github.com/guanzhi/GmSSL, 2024. Accessed: 2024-04-08.

[62] Botan-Project. Botan: Crypto and TLS for Modern C++. https://botan.randombit.net/, 2024. Accessed: 2024-04-08.

[63] Adalier M, Teknik A. Efficient and secure elliptic curve cryptography implementation of Curve P-256[J]. Workshop on Elliptic Curve Cryptography Standards, 2015. 446.

[64] 兰修文. ECC 计算算法的优化及其在 SM2 实现中的运用 [硕士学位论文], 2019.

[65] 何德彪, 陈泌文, 谢翔, et al. 一种适合 SM2 算法的快速模约减方法和介质 [专利], 2018.

[66] Donald E K. Art of computer programming, volume 2: Seminumerical algorithms. 第三版 [M]. Addison-Wesley Professional, 2014.

[67] Smith B. The Most Efficient Known Addition Chains for Field Element & Scalar Inversion for the Most Popular & Most Unpopular Elliptic Curves. https://briansmith.org/ecc-inversion-addition-chains-01, 2017.

[68] OpenSSL-Project. crypto/bn/bn_exp.c 中的 BN_mod_exp_mont_consttime 函数. Accessed: 2021-08-01.

[69] GmSSL-Project. crypto/ec/ecp_sm2z256.c 中的 ecp_sm2z256_mod_inverse 函数, 2021.

[70] Zhou L, Su C H, Hu Z, et al. Lightweight Implementations of NIST P-256 and SM2 ECC on 8-bit Resource-Constraint Embedded Device[J]. ACM Transactions on Embedded Computing Systems, 2019, 18(3):23:1–23:13.

[71] OpenSSL-Project. crypto/ec/asm 路径下的 ecp_nistz256-armv8.pl 脚本. Accessed: 2021-08-01.

[72] Booth A D. A signed binary multiplication technique[J]. The Quarterly Journal of Mechanics and Applied Mathematics, 1951, 4(2):236–240.

[73] Solinas J A. Efficient arithmetic on Koblitz curves[J]. Towards a Quarter-Century of Public Key Cryptography: A Special Issue of DESIGNS, CODES AND CRYPTOGRAPHY An International

Journal, 2000. 125–179.

[74] Percival C. Cache missing for fun and profit[J]. BSDCan Ottawa, 2005..

[75] Osvik D A, Shamir A, Tromer E. Cache attacks and countermeasures: the case of AES[J]. Cryptographers' track at the RSA conference, 2006. 1–20.

[76] OpenSSL-Project. crypto/bn 目录下 armv8-mont.S 中的 bn_mul_mont 函数, 编译时由 crypto/bn/asm/armv8-mont.pl 生成. Accessed: 2021-08-01.

[77] Düll M, Haase B, Hinterwälder G, et al. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers[J]. Designs, Codes and Cryptography, 2015, 77(2-3):493–514.

[78] Fujii H, Aranha D F. Curve25519 for the Cortex-M4 and Beyond[C]. Proceedings of Progress in Cryptology - LATINCRYPT 2017, volume 11368. Springer, 2017. 109–127.

[79] Hisil H, Egrice B, Yassi M. Fast 4 way vectorized ladder for the complete set of Montgomery curves[J]. IACR Cryptology ePrint Archive, 2020. 388.

[80] Nath K, Sarkar P. Efficient 4-Way Vectorizations of the Montgomery Ladder[J]. IEEE Transactions on Computers, 2022, 71(3):712–723.

[81] Cheng H, Fotiadis G, Groszschädl J, et al. Highly vectorized SIKE for AVX-512[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, 2022(2):41–68.

[82] Cheng H, Fotiadis G, Groszschädl J, et al. Batching CSIDH group actions using AVX-512[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, 2021(4):618–649.

[83] Faz-Hernández A. Artifact of the paper "High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions". https://github.com/armfazh/fld-ecc-vec, 2019. Accessed: 2023-07-01.

[84] Hisil H, Egrice B, Yassi M. Artifact of the paper "Fast 4 way vectorized ladder for the complete set of Montgomery curves". https://github.com/crypto-ninjaturtles/montgomery4x, 2020. Accessed: 2023-07-01.

[85] Nath K. Artifact of the paper "Efficient 4-Way Vectorizations of the Montgomery Ladder". https://github.com/kn-cs/vec-ladder, 2022. Accessed: 2023-07-01.

[86] Cheng H, Großschädl J, Tian J. Artifact of the paper "High-Throughput Elliptic Curve Cryptography Using AVX2 Vector Instructions". https://gitlab.uni.lu/APSIA/AVXECC, 2021. Accessed: 2023-07-01.

[87] Cheng H. Artifact of the paper "Highly vectorized SIKE for AVX-512". https://gitlab.uni.lu/APSIA/AVXSIKE, 2022. Accessed: 2023-07-01.

[88] Cheng H. Artifact of the paper "Batching CSIDH group actions using AVX-512". https://gitlab.uni.lu/APSIA/AVX-CSIDH, 2021. Accessed: 2023-07-01.

[89] Fog A. The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers. https://www.agner.org/optimize/microarchitecture.pdf, 2022. Accessed: 2023-07-01.

[90] Zinzindohoué J K, Bhargavan K, Protzenko J, et al. HACL*: A Verified Modern Cryptographic Library[C]. Proceedings of Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017. 1789–1806.

[91] Almeida J B, Barbosa M, Barthe G, et al. Jasmin: High-Assurance and High-Speed Cryptography[C]. Proceedings of Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017. 1807–1823.

[92] Erbsen A, Philipoom J, Gross J, et al. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises[C]. Proceedings of 2019 IEEE Symposium on Security and Privacy. IEEE, 2019. 1202–1219.

[93] Fu Y, Liu J, Shi X, et al. Signed Cryptographic Program Verification with Typed CryptoLine[C]. Proceedings of Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2019. 1591–1606.

[94] Solinas J A. Efficient Arithmetic on Koblitz Curves[J]. Designs, Codes and Cryptography, 2000, 19:195–249.

[95] Allen C, Dierks T. The TLS Protocol Version 1.0. RFC 2246, January, 1999. https://www.rfc-editor.org/info/rfc2246.

[96] Dierks T, Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, April, 2006. https://www.rfc-editor.org/info/rfc4346.

[97] GnuTLS-Project. GnuTLS library. https://gnutls.org/, 2024. Accessed: 2024-04-08.

[98] MbedTLS-Project. MbedTLS library. https://github.com/Mbed-TLS/mbedtls, 2024. Accessed: 2024-04-08.

[99] BoringTLS-Project. BoringTLS library. https://github.com/google/boringssl, 2024. Accessed: 2024-04-08.

[100] NSS-Project. NSS library. https://github.com/nss-dev/nss, 2024. Accessed: 2024-04-08.

[101] wolfSSL-Project. wolfSSL library. https://www.wolfssl.com/, 2024. Accessed: 2024-04-08.

[102] LibreSSL-Project. LibreSSL library. https://www.libressl.org/, 2024. Accessed: 2024-04-08.

[103] Tuveri N, Brumley B B. Start Your ENGINEs: Dynamically Loadable Contemporary Crypto[C]. Proceedings of 2019 IEEE Cybersecurity Development, SecDev 2019. IEEE, 2019. 4–19.

[104] Tuveri N, Brumley B B, Gridin I. Libsuola OpenSSL Engine. https://github.com/romen/libsuola, 2019. Accessed: 2023-07-01.

[105] Bernstein D J, Brumley B B, Chen M S, et al. OpenSSLNTRU: Faster post-quantum TLS key exchange[C]. Proceedings of 31st USENIX Security Symposium (USENIX Security 22), 2022. 845–862.

[106] OpenSSLNTRU-Project. Engntru OpenSSL Engine. https://opensslntru.cr.yp.to/engntru-20210608.tar.gz, 2021. Accessed: 2023-07-01.

[107] NLnetLabs. Unbound, a validating, recursive, and caching DNS resolver. https://nlnetlabs.nl/projects/unbound/about/, 2022. Accessed: 2023-07-01.

[108] Fog A. Test results for Broadwell and Skylake. https://www.agner.org/optimize/blog/read.php?i=415, December 26, 2015. Accessed: 2023-07-01.

[109] WikiChip. Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client), 2015. Accessed: 2023-07-01.

[110] Intel-PMU-Project. pmu-tools. https://github.com/andikleen/pmu-tools, 2023. Accessed: 2023-07-01.

[111] Yasin A. A top-down method for performance analysis and counters architecture[C]. Proceedings of 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2014. 35–44.

[112] Unbound-Project. Howto Statistics. https://www.nlnetlabs.nl/documentation/unbound/howto-statistics/, 2022. Accessed: 2023-07-01.

[113] Larabel M. AVX / AVX2 / AVX-512 Performance + Power On Intel Rocket Lake. https://www.phoronix.com/review/rocket-lake-avx512/6, 2021. Accessed: 2023-07-01.

[114] OpenSSL-Project. X25519 x64 assembly language implementation. https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/ec/asm/x25519-x86_64.pl, 2018. Accessed: 2023-07-01.

[115] OpenSSL-Project. Ed25519 C language implementation. https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/ec/curve25519.c, 2016. Accessed: 2023-07-01.

[116] Bernstein D J. Analyzing the complexity of reference post-quantum software[J]. Cryptology ePrint Archive, 2023..

[117] OpenSSL-Project. OpenSSL s_server tool. https://beta.openssl.org/docs/manmaster/man1/openssl-s_server.html, 2022. Accessed: 2023-07-01.

[118] OpenSSLNTRU-Project. tls_timer tool. https://opensslntru.cr.yp.to/tls_timer-20210608.tar.gz, 2021. Accessed: 2023-07-01.

[119] Huang J, Zhang J, Zhao H, et al. Improved Plantard Arithmetic for Lattice-based Cryptography[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, 2022(4):614–636.

[120] Alkim E, Ducas L, Pöppelmann T, et al. Post-quantum Key Exchange - A New Hope[C]. Proceedings of 25th USENIX Security Symposium, USENIX Security 16. USENIX Association, 2016. 327–343.

[121] Alkim E, Bilgin Y A, Cenk M, et al. Cortex-M4 optimizations for {R, M} LWE schemes[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020, 2020(3):336–357.

[122] Greconici D O C, Kannwischer M J, Sprenkels D. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, 2021(1):1–24.

[123] Abdulrahman A, Hwang V, Kannwischer M J, et al. Faster Kyber and Dilithium on the Cortex-M4[C]. Proceedings of Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, volume 13269. Springer, 2022. 853–871.

[124] Plantard T. Efficient Word Size Modular Arithmetic[J]. IEEE Transactions on Emerging Topics in Computing, 2021, 9(3):1506–1518.

[125] SiFive. SiFive FE310-G002 Manual. https://sifive.cdn.prismic.io/sifive/b56b304f-cd2d-421b-9c14-6b35c33f172e_fe310-g002-manual-v1p4.pdf, 2021. Accessed: 2023-07-01.

[126] Canaan-Inc. Canaan Kendryte K230 documentation. https://github.com/kendryte/k230_docs/blob/main/README_en.md, 2024. Accessed: 2024-07-01.

[127] T-Head. XuanTie-C908-UserManual. https://www.xrvm.com/product/xuantie/C908, 2023. Accessed: 2023-10-01.

[128] RISC-V Foundation. RISC-V V Vector Extension Version 1.0-rc1-20210608. https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf, 2021.

[129] RISC-V Foundation. RISC-V Bit-Manipulation ISA-extensions Version 1.0.0-2021-06-12. https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf, 2021.

[130] Fog A. Test programs for measuring clock cycles and performance monitoring. https://agner.org/optimize/testp.zip, 2023.

[131] Abel A, Reineke J. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions

on Intel Microarchitectures[C]. Proceedings of ASPLOS, New York, NY, USA: ACM, 2019. 673–686.

[132] Huang J, Zhao H, Zhang J, et al. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices[J]. IEEE Transactions on Information Forensics and Security, 2024..

[133] Huang J, Adomnicai A, Zhang J, et al. Revisiting Keccak and Dilithium Implementations on ARMv7-M[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2024, 2024(2):1–24.

[134] PQM3-Project. PQM3: Post-quantum crypto library for the ARM Cortex-M3. https://github.com/mupq/pqm3. Accessed: 2023-07-01.

[135] Abdulrahman A, Chen J, Chen Y, et al. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, 2022(1):127–151.

[136] Lyubashevsky V, Seiler G. NTTRU: Truly Fast NTRU Using NTT[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019, 2019(3):180–201.

[137] 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[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2021, 2021(2):159–188.

[138] Abdulrahman A, Hwang V, Kannwischer M J, et al. Faster Kyber and Dilithium on the Cortex-M4[C]. Proceedings of Applied Cryptography and Network Security, ACNS 2022, volume 13269. Springer, 2022. 853–871.

[139] Abdulrahman A, Chen J, Chen Y, et al. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2022, 2022(1):127–151.

[140] Karmakar A, Mera J M B, Roy S S, et al. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM[J]. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018, 2018(3):243–266.

[141] Liang Z, Fang B, Zheng J, et al. Compact and efficient KEMs over NTRU lattices[J]. Computer Standards & Interfaces, 2024, 89:103828.

[142] Lu X, Liu Y, Zhang Z, et al. LAC: Practical ring-LWE based public-key encryption with byte-level modulus[J]. Cryptology ePrint Archive, 2018..

[143] Lorenser T. The DSP capabilities of ARM Cortex-M4 and Cortex-M7 Processors[J]. ARM White Paper, 2016, 29.

[144] Bertoni G, Daemen J, Hoffert S, et al. Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf, 2012. Accessed: 2023-05-26.

[145] Becker H, Kannwischer M J. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64[C]. Proceedings of Progress in Cryptology – INDOCRYPT 2022, Cham: Springer International Publishing, 2022. 272–293. https://eprint.iacr.org/2022/1243.

[146] Stoffelen K. Efficient Cryptography on the RISC-V Architecture[C]. Proceedings of Progress in Cryptology - LATINCRYPT 2019, volume 11774. Springer, 2019. 323–340.

[147] Lenngren E. AArch64 optimized implementaton for X25519. https://github.com/Emill/X25519-AArch64, 2019.

致　　谢

在本论文的完成过程中，我深知没有许多人在背后默默付出与支持，我无法顺利走到今天。在此，我要特别感谢对我提供过帮助的人。

我要感谢我的老师、同学和朋友们，在学术交流和实习经历中，您们给予了我丰富的视角和不懈的支持。每一次讨论与交流都让我受益匪浅，让我更加坚定了自己的研究方向和学术信念。

我要特别感谢我的家人，感谢你们无条件的理解、支持与鼓励。无论在学术上遇到多大的困难，您们的关爱始终是我最强大的精神支柱，给了我坚持下去的勇气和力量。

最后，我也要感谢我的合作者们，在研究过程中您们的共同努力与交流让我受益良多。我们的合作不仅提升了学术研究的深度与广度，也让我更加深刻地体会到团队合作的力量。

再次感谢所有帮助过我的人，正是有了你们的支持与鼓励，我才能够完成这篇论文，走到今天。

在学期间的研究成果及发表的学术论文

攻读博士学位期间发表（录用）论文情况

1. **Jipeng Zhang**, Yuxing Yan, Junhao Huang, Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. CHES 2025. **CCF-B** 会议 & 密码工程顶会.
2. **Jipeng Zhang**, Junhao Huang, Lirui Zhao, Donglong Chen, Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. Usenix Security 2024. **CCF-A** 会议 & 安全四大顶会. **Distinguished Paper Award**.
3. **张吉鹏**, 黄军浩, 等人. 面向移动设备的国密 SM2 高效实现研究. 电子学报 2023. **CCF-T1**.
4. **Jipeng Zhang**, Junhao Huang, et al. Time-memory Trade-offs for Saber+ on Memory-constrained RISC-V Platform. IEEE Transactions on Computers 2022. **CCF-A** 期刊. SCI 二区.
5. **Jipeng Zhang**, et al. An Efficient and Scalable Sparse Polynomial Multiplication Accelerator for LAC on FPGA. IEEE ICPADS 2020. **CCF-C** 会议.
6. Junhao Huang, Alexandre Adomnicăi, **Jipeng Zhang**, et al. Revisiting Keccak and Dilithium Implementations on ARMv7-M. CHES 2024. **CCF-B** 会议 & 密码工程顶会.
7. Junhao Huang, Haosong Zhao, **Jipeng Zhang**, et al. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. IEEE Transactions on Information Forensics & Security 2024. **CCF-A** 期刊. SCI 一区.
8. Xuan Yu, **Jipeng Zhang**, Junhao Huang, et al. Multi-way High-throughput Implementation of Kyber. ISC 2024. **CCF-C** 会议.
9. Junhao Huang, **Jipeng Zhang**, et al. Improved Plantard arithmetic for lattice-based cryptography. CHES 2022. **CCF-B** 会议 & 密码工程顶会.
10. Lirui Zhao, **Jipeng Zhang**, et al. Efficient Implementation of Kyber on Mobile Devices. IEEE ICPADS 2021. **CCF-C** 会议.