Software Implementations of Polynomial Multiplications for Lattice-Based Cryptosystems

Vincent Bert Hwang

Software Implementations of Polynomial Multiplications for Lattice-Based Cryptosystems

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus **TODO:** name ,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

TODO: date

TODO: time

door

Vincent Bert Hwang

geboren op **TODO:** date
te **TODO:** location

Promotor

Prof. dr. Peter Schwabe

Copromotor

Prof. dr. Bo-Yin Yang
Academia Sinica, Taiwan

Manuscriptcommissie

Prof. dr. Lejla Batina

Dr. ing. Thomas Pöppelmann
Infineon Technologies AG, Duitsland

Prof. dr. ing. Tim Güneysu
Ruhr-Universität Bochum, Duitsland

Prof. dr. ir. Ingrid Verbauwhede
Katholieke Universiteit Leuven, België

Dr. Francisco Rodríguez-Henríquez
Technology Innovation Institute, Verenigde Arabische Emiraten

Contents

List of Algorithms

Chapter 10: General Guide for Optimizing Transformations 161

Chapter 12: Dilithium 183

Chapter 13: Kyber **199**

Chapter 14: NTRU **217**

Chapter 15: NTRU Prime **231**

Chapter 16: Saber **251**

Chapter A: On Formal Verification **293**

Chapter B: Large Integer Multiplications **353**

List of Figures

List of Listings

List of Tables

Chapter 13: Kyber **199**

Chapter 14: NTRU 217

Chapter 15: NTRU Prime 231

Chapter 16: Saber 251

Chapter A: On Formal Verification 293

Chapter B: Large Integer Multiplications 353

Chapter 1

Introduction

> *The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.*
>
> *Donald E. Knuth, Fundamental Algorithms*
> *Volume 1 of The Art of Computer Programming*

1.1 Subject of Research

Cryptography studies how secure communications can be established under various adversarial models and real-world attacks.

Symmetric cryptography and asymmetric cryptography. There are two major lines of cryptography: symmetric cryptography and asymmetric cryptography. In symmetric cryptography, all the parties hold the same secret key and transform the plaintext into ciphertext and ciphertext into plaintext with the secret key or procedures derived from the secret key. This allows parties sharing a secret key to transfer large chunk of confidential data through insecure channels by transforming plaintext back-and-forth. Asymmetric cryptography removes the need of sharing the secret key among parties.

Post-quantum cryptosystems. Although large-scale cryptanalytic-relevant quantum computers are not yet available, some secret information nowadays is susceptible to attacks by future quantum computers. For an encryption cryptosystem, an attacker stores the public information, and extracts the secret key and decrypts the ciphertext once large-scale quantum computers are available. For symmetric cryptography, Grover's algorithm reduces the brute-force search bit-complexity by half in quantum computation model [Gro96], and doubling the key size is sufficient to achieve comparable security. However, things are quite different for asymmetric cryptography. Popular approaches such as Rivest–Shamir–Adleman relying on the hardness of integer factorization and elliptic-curve cryptography relying on the hardness of elliptic-curve discrete logarithm are broken by Shor's algorithms in quantum computation model [Sho97a]. Post-quantum cryptography studies cryptosystems relying on hard problems that are believed to be secure against quantum computers. There are five major lines of post-quantum cryptosystems in post-quantum cryptography: lattice-based cryptosystems, multivariate cryptosystems, hash-based cryptosystems, code-based cryptosystems, and isogeny-based cryptosystems.

NIST Post-Quantum Cryptography Standardization. At PQCrypto 2016, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography Standardization with multi-round evaluation, soliciting post-quantum cryptography schemes implementing key-encapsulation mechanisms (KEMs) and digital signatures. NIST received 82 candidates, and 69 candidates were regarded complete and participated the first-round evaluation. Lattice-based cryptosystems are among the most popular constructions due to their well-balanced performance in terms of the amount of data to be transmitted and the processing time. Indeed, NIST advanced 26 submissions to round two on January 30th, 2019, and among the 26 candidates, 12 were lattice-based, 7 were code-based, 1 was hash-based, 4 were multivariate-based, 1 was super-singular-isogeny-based, and 1 was zero-knowledge-proof-based [AASA+19]. NIST advanced 15 candidates to round three on July 22nd, 2020 [AASA+20] and 7 of them were lattice-based cryptosystems. On July 5th, 2022, NIST selected 5 cryptosystems as the first group of cryptosystems for standardization [AAC+22] and 3 were lattice-based. NIST also moved 4 candidates to round four and one of the candidates was broken in round four. On March 11th, 2025, NIST selected a candidate for standardization from round four [ABC+25].

1.1.1 Lattice-Based Cryptosystems and Platforms

In a typical lattice-based cryptosystem, one of the computational bottlenecks is polynomial arithmetic such as polynomial multiplications over integer rings, matrix multiplications over integer rings, and matrix-to-vector multiplications over polynomial rings. This thesis reviews and improves the software implementations of polynomial arithmetic in five out of the seven lattice-based cryptosystems in the third round of NIST PQC Standardization on five platforms. For the lattice-based cryptosystems, this thesis looks into Dilithium, Kyber, NTRU, NTRU Prime, and Saber (in alphabetical order). As for the platforms, this thesis focuses on Cortex-M3 implementing Armv7-M, Cortex-M4 implementing Armv7E-M, Cortex-A72 implementing Armv8.0-A, Firestorm implementing Armv8.4-A, and Haswell implementing x86-64 with AVX2. See Tables 1.1 and 1.2 for an overview.

Table 1.1: Scope of this thesis on Cortex-M3 and Cortex-M4.

Cryptosystem	Cortex-M3	Cortex-M4
	Armv7-M	Armv7E-M
Dilithium	Reviewing/improving.	Reviewing/improving.
Kyber	Reviewing/improving.	Reviewing/improving.
NTRU	-	Reviewing.
NTRU Prime	-	Reviewing.
Saber	Reviewing/improving.	Reviewing.

Table 1.2: Scope of this thesis on Cortex-A72, Firestorm, and Haswell.

Cryptosystem	Cortex-A72/Firestorm	Haswell
	Armv8.0-A/Armv8.4-A	x86-64 with AVX2
Dilithium	Reviewing/improving.	Reviewing.
Kyber	Reviewing/improving.	Reviewing/improving.
NTRU	Reviewing/improving.	Reviewing.
NTRU Prime	Reviewing/improving.	Reviewing/improving.
Saber	Reviewing.	Reviewing.

1.1.2 Assembly

We specifically focus on assembly-optimized implementations for the performance-critical polynomial arithmetic in lattice-based cryptosystems. Natural concerns are the engineering effort, maintainability, and portability of assembly-optimized implementations. Maintainability heavily depends on the programming practice and varies a lot between programmers. It is true that programming directly in assembly amounts to huge engineering effort and suffers from portability across different architectures. However, there are several issues if we do not have fine-grained control on the assembly, and the benefit of programming in assembly payoff if we focus on the performance-critical component. We outline below the implementation challenges.

Secret-independent execution time. A function has secret-independent execution time[1] if the execution time is independent from the secret inputs. There are at least three possible sources of secret-dependent execution time – conditional branches, memory access patterns, and assembly instructions. If the condition of a conditional branch is evaluated differently for different secret inputs, then the computational flows will be different and often lead to different execution time. If the memory location to be accessed varies for different secret inputs, we could possibly hit cache misses for some inputs and end up spending more time loading from memory for some secret inputs. At the assembly-instruction level, there are some instructions whose execution time depend on the secret inputs. When any of these occurs, the attacker learns some information on the secret inputs. A straightforward countermeasure is to implement functions with secret inputs with carefully selected assembly instructions, so compilers have no chances to replace the secret-independent computations with the often faster secret-dependent computations.

Same instructions, different platforms. When executing a function, the optimization strategies might be quite different on different platforms. For example, while copying an array of elements to another array on Cortex-M4 and Cortex-M7 both implementing the instruction set architecture Armv7E-M, grouping the loads together and the stores together is faster on a Cortex-

---

[1] Secret-independent execution time is also called constant-time. However, the term constant-time does not necessarily mean constant execution time. Computations on the public do not need to run in constant execution time. If the overall computation has variable execution time and the computations on secret data has constant execution time, we also call it constant-time. However, the term constant-time does not align well with the common usage of constant-time and this thesis deliberately chooses secret-independent execution time to avoid this.

M4 processor while interleaving the loads and stores is faster on a Cortex-M7 processor. Programing in assembly gives us full control over instruction scheduling.

Same platform, compiler, and code, different compilation flags. On the same platform and with the same compiler, the same program might be compiled into different assembly programs if we change the compilation flags. For example, the C operator % is compiled into division, multiplication, and subtraction with optimization flag `-O0`, and sometimes into a string of instructions excluding divisions with `-O3` [Dan24]. As the execution time of division instructions often depend on the inputs, it is best to avoid them. A straightforward way is to replace the % operator with a string of assembly instructions without divisions.

Support of vector instructions. Vector instructions are now commonly implemented on high-performance processors. From the programming point of view, a batch of arithmetic of the same kind is packed as a single instruction, allowing us to issue a batch of computations with small instruction-decoding bandwidth. While auto-vectorization is a common compiler optimization, programming in assembly ensures the desired vectorization.

Same platform and compiler, different code structure. Intrinsics are architecture-specific functions/macros allowing programmers to access the low-level special assembly instructions in a high-level programming language. However, the mapping between intrinsics and assembly instructions are not necessarily uniquely determined. We illustrate an example with matrix multiplications. Let $\mathtt{I}, \mathtt{J}, \mathtt{K}$ be positive integers that are multiples of four. Given matrices $A \in M_{\mathtt{I}\times\mathtt{J}}(\mathbb{Z}_{2^{32}}), B \in M_{\mathtt{J}\times\mathtt{K}}(\mathbb{Z}_{2^{32}}), C \in M_{\mathtt{I}\times\mathtt{K}}(\mathbb{Z}_{2^{32}})$, we wish to compute $AB + C$ over $\mathbb{Z}_{2^{32}}$. The computing task is named as "matrix multiplication with accumulation of dimension $\mathtt{I} \times \mathtt{J} \times \mathtt{K}$." Listing 1.1 is a C implementation iterating in the order i, j, k with $\mathbb{Z}_{2^{32}}$ encoded as `int32_t`.

Listing 1.1: Matrix multiplication with accumulation of dimension $\mathtt{I} \times \mathtt{J} \times \mathtt{K}$.

```
void matmla_ijk_int32(int32_t *C, const int32_t *A, const
    int32_t * B){
    // A: I by J
    // B: J by K
    // C: I by K
    for(size_t i = 0; i < I; i++){
        for(size_t j = 0; j < J; j++){
```

```
                for(size_t k = 0; k < K; k++){
                    C[i * K + k] += A[i * J + j] * B[j * K + k];
                }
            }
        }
    }
```

On Armv8-A Neon, a vector instruction set comes with the Armv8-A instruction set architecture, there are vector loads, stores, and multiplications with accumulations. There are two kinds of multiplications: vector-by-vector and vector-by-scalar. A vector-by-vector multiplication component-wisely multiplies the elements of two vectors, and a vector-by-scalar multiplication multiplies the scalar operand to each of the elements of the vector operand. We again iterate in the order i, j, k and rewrite the inner loops with vector loads, stores, and vector-by-scalar multiplications in intrinsics as shown in Listing 1.2. On an Apple M1 Pro running `Sonoma 14.6.1`, the inner loops are compiled into vector duplications and vector-by-vector multiplications in assembly with the compiler `Apple clang version 15.0.0` and optimization flags `-O3 -march=native`. If we instead iterate in the order i, k, j as shown in Listing 1.3, the inner loops are compiled into vector-by-scalar multiplications as expected. Programming in assembly gives us full control on instruction selections and we do not need to rely on compilers to select the desired instructions.

Listing 1.2: Matrix multiplication with accumulation of dimension I × J × K iterating in the order i, j, k with Armv8-A Neon intrinsics in C.

```
void matmla_ijk_lane_int32(int32_t *C, const int32_t *A, const
    int32_t * B){
    int32x4_t cx4, ax4, bx4[4];
    for(size_t i = 0; i < I; i++){
        for(size_t j = 0; j < J; j += 4){
            ax4 = vld1q_s32(A + i * J + j);
            for(size_t k = 0; k < K; k += 4){

                bx4[0] = vld1q_s32(B + (j + 0) * K + k);
                bx4[1] = vld1q_s32(B + (j + 1) * K + k);
                bx4[2] = vld1q_s32(B + (j + 2) * K + k);
                bx4[3] = vld1q_s32(B + (j + 3) * K + k);

                /* Below are compiled into duplications and
                    vector-by-vector multiplications. */
                cx4 = vmlaq_laneq_s32(cx4, bx4[0], ax4, 0);
                cx4 = vmlaq_laneq_s32(cx4, bx4[1], ax4, 1);
                cx4 = vmlaq_laneq_s32(cx4, bx4[2], ax4, 2);
                cx4 = vmlaq_laneq_s32(cx4, bx4[3], ax4, 3);
```

```
                vst1q_s32(C + i * K + k, cx4);

            }

        }
    }
}
```

Listing 1.3: Matrix multiplication with accumulation of dimension `I` × `J` × `K` iterating in the order i, k, j with Armv8-A Neon intrinsics in C.

```
void matmla_ikj_lane_int32(int32_t *C, const int32_t *A, const
    int32_t * B){
    int32x4_t cx4, ax4, bx4[4];
    for(size_t i = 0; i < I; i++){
        for(size_t k = 0; k < K; k += 4){
            cx4 = vld1q_s32(C + i * K + k);
            for(size_t j = 0; j < J; j += 4){

                ax4 = vld1q_s32(A + i * J + j);
                bx4[0] = vld1q_s32(B + (j + 0) * K + k);
                bx4[1] = vld1q_s32(B + (j + 1) * K + k);
                bx4[2] = vld1q_s32(B + (j + 2) * K + k);
                bx4[3] = vld1q_s32(B + (j + 3) * K + k);

                /* Below are compiled into vector-by-scalar
                    multiplications. */
                cx4 = vmlaq_laneq_s32(cx4, bx4[0], ax4, 0);
                cx4 = vmlaq_laneq_s32(cx4, bx4[1], ax4, 1);
                cx4 = vmlaq_laneq_s32(cx4, bx4[2], ax4, 2);
                cx4 = vmlaq_laneq_s32(cx4, bx4[3], ax4, 3);

            }
            vst1q_s32(C + i * K + k, cx4);
        }

    }
}
```

1.2 Contributions of This Thesis

1.2.1 Microcontrollers

On microcontrollers, the first two publications were published when I was an undergraduate student at National Taiwan University. According to the regulations of National Taiwan University, only publications published after the start date of a master's program are allowed to be included in the master's thesis. Therefore, the first two publications were not included in my master thesis and they are included as contributions of this thesis.

Polynomial multiplications for NTRU Prime on Cortex-M4. The first contribution targets the polynomial multiplications for NTRU Prime on Cortex-M4, and is based on the following published work.

> Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime: Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2020. Paper. Artifact. Talk. Slides. IACR ePrint. Reference [ACC+20].
>
> **Summary:** We proposed three assembly-optimized NTT-based implementations for the polynomial multiplication in NTRU Prime on Cortex-M4. I was deeply involved in the Good–Thomas implementation.

We explored several polynomial multiplication strategies, including Good–Thomas FFT, Rader-17 FFT, and Cooley–Tukey FFT, Toom–Cook, and several implementation techniques, such as the uses of floating-point registers as low-latency cache and multi-layer butterflies to reduce the memory operations. We chose a 32-bit NTT-friendly modulus and applied Good–Thomas and Cooley–Tukey FFTs with the 32-bit Montgomery multiplication using 32-bit multiplication instructions. We also proposed three implementations exploiting the 16-bit Digital Signal Processing multiplication instructions: Toom–Cook, mixed-radix with small radices, and Rader-17. I was deeply involved in the development of Good–Thomas FFT and Cooley–Tukey FFT implementations.

Polynomial multiplications for NTRU, Saber, and LAC on Cortex-M4 and Skylake. The second contribution targets the polynomial multiplications for NTRU, Saber, and LAC on Cortex-M4 and Skylake, and is based on the following published work.

> Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT

> Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. Paper. Artifact. Talk. Slides. IACR ePrint.
> Reference [CHK+21].
>
> **Summary:** We migrated and implemented the Good–Thomas approach by [ACC+20] to all the parameter sets of NTRU on Cortex-M4. Additionally, we also optimized Saber on Cortex-M4 and NTRU and Saber on Skylake with AVX2. I was deeply involved in the Cortex-M4 implementations.

We applied NTTs to the NTT-unfriendly rings in NTRU, Saber, and LAC on Cortex-M4 and Skylake, and improved the prior art significantly. I was deeply involved in the Cortex-M4 implementations.

Polynomial multiplications for Dilithium and Kyber on Cortex-M4. The third contribution targets the polynomial multiplications for Dilithium and Kyber on Cortex-M4, and is based on the following published work.

> Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *International Conference on Applied Cryptography and Network Security*, pages 853–871. Springer. 2022. Paper. Artifact. IACR ePrint. Reference [AHKS22].
>
> **Summary:** We revisited the optimization of Dilithium and Kyber on Cortex-M4. We incorporated assembly optimizations from prior Cortex-M4 works [ACC+20, CHK+21, ACC+21] and Cortex-A72 work [BHK+21], and optimized the challenge polynomial multiplications in Dilithium. I proposed the Fermat number transform for the challenge polynomial multiplication in Dilithium, an optimization for the base multiplication in Kyber, and wrote the corresponding part of the paper with other coauthors.

We revisited the assembly-optimized implementations of Dilithium and Kyber on Cortex-M4. In particular, we used floating-point registers as low-latency cache [ACC+20], implemented the Barrett reduction with wide multiplication instructions [ACC+20], and applied the asymmetric multiplication [BHK+21] with dual multiplication instructions. We also proposed several optimization techniques for the challenge polynomial multiplications in the rejection loop of Dilithium, such as choosing new NTT-friendly 16-bit moduli and applying

Fermat number transform. I proposed the use of Fermat number transform, provided some insights on the dual multiplication instructions, and reviewed the implementations.

Polynomial multiplications for Dilithium and Saber on Cortex-M3 and 8-bit AVR. The fourth contribution targets the polynomial multiplications for Dilithium and Saber on Cortex-M3 and 8-bit AVR, and is based on the following published works.

> Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Barrett Multiplication for Dilithium on Embedded Devices. Cryptology ePrint Archive, Paper 2023/1955, 2023. Paper. Extended to [HKS24]. Reference [HKS23].
>
> **Summary:**
>
> Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying Polynomials without Powerful Multiplication Instructions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):160–202, 2024. Paper. Artifact. Slides. IACR ePrint. Reference [HKS24].
>
> **Summary:** We optimized Dilithium and Saber on Cortex-M3 and 8-bit AVR. For the Dilithium coefficient ring modulo a prime, we generalized Barrett multiplication and proposed several versions suitable for multi-limb arithmetic on platforms without powerful multiplication instructions. For the power-of-two-coefficient ring case, we showed that Nussbaumer performs the best if there are no precision issues. I proposed the ideas, implemented the Cortex-M3 optimizations, and wrote the corresponding parts of the paper.

We generalized the notion of integer approximations, introduced several variants of Barrett multiplications trading the quality of modular multiplications with the computational efficiency, and applied the ideas to the 32-bit Dilithium $\mathtt{NTT}/\mathtt{NTT}^{-1}$ on Cortex-M3 and 8-bit AVR. We also proposed the uses of Nussbaumer+Toeplitz-TC over $\mathbb{Z}_{2^k}$ for the challenge polynomial multiplications in Dilithium and the matrix-vector multiplications in Saber. The Nussbaumer+Toeplitz-TC approach is the fastest on Cortex-M3, and the NTT approach is the fastest on 8-bit AVR due to the rather large k in Saber with 8-bit arithmetic. I proposed the ideas, implemented the Cortex-M3 implementations, and wrote the paper except for the 8-bit AVR parts.

1.2.2 High Performance Processors with Vector Instructions

Improved instruction scheduling for the Armv8-A Neon `NTT`/$\texttt{NTT}^{-1}$. The fifth contribution improves the instruction scheduling of the Armv8-A Neon `NTT`/$\texttt{NTT}^{-1}$ and was not publicly available due to the time constraint at the time of the submission of the coauthored prior work [BHK+21]. The work [BHK+21] was already included as a contribution of my master's thesis and is not a contribution of this thesis.

Polynomial multiplications for NTRU with Armv8-A Neon. The sixth contribution targets the polynomial multiplications for NTRU with Armv8-A Neon, and is based on the following published work.

> Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU. In Anupam Chattopadhyay, Shivam Bhasin, Stjepan Picek, and Chester Rebeiro, editors, *Progress in Cryptology – INDOCRYPT 2023*, pages 177–196. Springer, 2024. Paper. Artifact. Slides. IACR ePrint. Reference [CCHY24].
>
> **Summary:** We optimized NTRU on Cortex-A72 with Armv8-A Neon. We improved the design choice of Toom–Cook and implemented the Toeplitz matrix-vector product approach for the polynomial multiplications in NTRU. I proposed the improvement of Toom–Cook, integrated the polynomial multipliers by coauthors, implemented the then-not-yet-deployed obvious optimizations, wrote the non-implementation part of the paper, and refined the implementation part of the paper.

We investigated several non-NTT-based approaches over $\mathbb{Z}_{2^k}$ for NTRU with Armv8-A Neon. We proposed a Toom–Cook approach built upon the Toom-5 requiring inverses of powers of two up to 2^3, performed some memory optimizations during the interpolations of Toom–Cook, and demonstrated the benefit of the Toeplitz-based approach with the vector-by-scalar multiplication instructions. I proposed the Toom-5 and the resulting decomposition strategy, rephrased the relation of Toeplitz matrix-vector products and polynomial multiplications with dual modules, and wrote most of the paper.

Polynomial multiplications for NTRU Prime with Armv8-A Neon. The seventh contribution targets the polynomial multiplications for NTRU Prime with Armv8-A Neon, and is based on the following published work.

> Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. In *International Conference on Applied Cryptography and Network Security*, pages 24–46. Springer, 2024. Paper. Artifact. Slides. IACR ePrint. Reference [HLY24].
>
> **Summary:** We optimized NTRU Prime on Cortex-A72 with Armv8-A Neon. We explored Schönhage and Bruun's FFTs, integrated them with Good–Thomas and Rader's FFTs, and proposed two approaches for multiplying polynomials in NTRU Prime. Our approaches significantly reduced the number of small-dimensional polynomial multiplications while vectorizing for NTRU Prime. I proposed the optimizations, implemented the Bruun's FFT and the fastest approach, and wrote the paper.

We investigated several FFTs, such as Schönhage's, Nussbaumer's, Good–Thomas, Cooley–Tukey, and Bruun's FFTs, and proposed two transformations amenable for vectorization with Armv8-A Neon while reducing the number of resulting small-dimensional polynomial multiplications compared to prior vectorization work [BBCT22]. I proposed the ideas, implemented the fastest one, and wrote most of the paper.

Further vectorization for NTRU Prime with Armv8-A Neon and AVX2. The eighth contribution furthers the polynomial multiplications for NTRU Prime with Armv8-A Neon and AVX2, and is based on the following published work.

> Vincent Hwang. Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime. In *Australasian Conference on Information Security and Privacy*, pages 84–102. Springer, 2024. Paper. Artifact. Slides. IACR ePrint. Reference [Hwa24c].
>
> **Summary:** We furthered the NTRU Prime optimizations by [HLY24] and optimized on Cortex-A72 with Armv8-A Neon and on Haswell with AVX2. The paper systematically reviewed the notion of vectorization on various platforms, and replaced Rader's FFT with truncated Rader's FFT. I proposed the ideas, implemented the optimizations, and wrote the paper.

This work refined the vectorized polynomial multiplications for NTRU Prime with Armv8-A Neon and AVX2. The work replaced Rader's FFT with truncated Rader's FFT and applied Toeplitz matrix-vector multiplications to the

Armv8-A Neon implementation. I proposed the ideas, implemented the optimizations, and wrote the paper.

Improved bit-sliced polynomial inversions over $\mathbb{Z}_3$ in NTRU and NTRU Prime. The ninth contribution picked up a missing optimization for the bit-sliced polynomial inversion over $\mathbb{Z}_3$ in NTRU and NTRU Prime with Armv8-A Neon. The optimization was based on the existing AVX2 implementations found in [BBC$^+$20, CDH$^+$20].

1.2.3 Quotient

Improved Kyber compressions with Armv7-M, Armv7E-M, Armv8-A, and AVX2. The tenth contribution improves the compressions in Kyber with Armv7-M, Armv7E-M, Armv8-A, and AVX2, and is included in the following submission accepted by IEEE Security and Privacy 2025.

> Gilles Barthe, Gustavo Xavier Delerue Marinho Alves, Hugo Pacheco, José Bacelar Almeida, Luís Esquível, Manuel Barbosa, Peter Schwabe, Pierre-Yves Strub, Tiago Oliveira, and Vincent Hwang. Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3526–3544. IEEE Computer Society, 2025. Paper. IACR ePrint. Reference [AAB$^+$25].
>
> **Summary:** We combined deductive reasoning and circuit-based reasoning in EasyCrypt, verified the AVX2-optimized rejection sampling in Kyber for the first time and the equivalences of the optimized implementations of the compression functions, and simplified the verification of the AVX2-optimized Keccak permutation. I proposed the optimizations for the compression functions and wrote the corresponding part in the paper.

We implemented a hybrid formal verification approach combining the high-level deductive reasoning and the circuit-based reasoning, and verified the correctness of the rejection sampling and compressions in Armv7E-M and AVX2. I proposed the improvements of compressions, implemented the compressions with Armv7-M, Armv7E-M, Armv8-A, and AVX2, and wrote the corresponding section.

1.2.4 Survey

A survey of polynomial multiplications in Dilithium, Kyber, NTRU, NTRU Prime, and Saber. The eleventh contribution surveys several techniques of polynomial multiplications in Dilithium, Kyber, NTRU, NTRU Prime, and Saber, and is based on the following published work.

> Vincent Hwang. A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems. *IACR Communications in Cryptology*, 1(2), 2024. Paper. IACR ePrint. Reference [Hwa24a].
>
> **Summary:** This paper reviewed several techniques for multiplying polynomials in polynomial rings of the form $\mathbb{Z}_q[x]/\langle x^n - \alpha x - \beta\rangle$ in practice. There are three emphases: modular arithmetic, homomorphisms, and vectorization. I wrote the paper.

I wrote the paper.

1.3 Research Data Management

The implementations that are either counted as contributions of this thesis or marked as "Benchmark of this thesis" are included in the repository `https://github.com/vincentvbh/PhD_thesis_MPI-SP` and the archive `https://doi.org/10.5281/zenodo.15847923`. Numbers marked as "Benchmark of this thesis" are not necessary contributions of this thesis. If the cited work is a contribution of this thesis, then the number is a contribution of this thesis; if the cited work is not a contribution of this thesis, then the number is not a contribution of this thesis. Re-benchmarking existing works, whether contributions of this thesis or not, is necessary in several cases due to the inconsistency on the versions of cryptosystems, platform settings, and shared subroutines that are out of the scope of this thesis. Numbers marked as "This thesis" are benchmark of unpublished improvement are contributions of this thesis See Appendix C for further information.

1.4 Related Survey Works

Other than this thesis, there are several survey works reviewing polynomial multiplications. Please refer to [Nus82, DV90, Ber01, Ber08b, Hwa22, LZ22].

1.5 Additional Publications

This section outlines publications that are not included in the main body of the thesis.

1.5.1 Additional Publications During the PhD Studies

Formal verification of emulated floating-point arithmetic. The following published work studies the formal verification of emulated floating-point arithmetic, and is included in Appendix A.1.

> Vincent Hwang. Formal Verification of Emulated Floating-Point Arithmetic in Falcon. In *International Workshop on Security*, pages 125-141. Springer, 2024. Paper. Artifact. Slides. IACR ePrint. Reference [Hwa24b].
>
> **Summary:** This paper verified the functional equivalences of the software-emulated floating-point additions/multiplications and the range of the intermediate floating-point values of the FFT computations in the signature generation of Falcon with the domain-specific language CryptoLine. I proposed the ideas, carried out the verification, and wrote the paper.

This paper pointed out a discrepancy between the emulated floating-point multiplication in the submission package of the digital signature Falcon [PFH+20] and the claimed behavior. With CryptoLine, the paper also modeled the floating-point arithmetic, showed that the discrepancy does not affect the correctness of the FFT in the signature generation of Falcon, and demonstrated the equivalences between emulated floating-point arithmetic. I identified the discrepancy, proposed the ideas, exercised the verification, and wrote the paper.

1.5.2 Additional Publications Prior to the PhD Studies

There are five published works prior to PhD studies that are not included in the main body of the thesis.

Formal verification of NTTs. The following published work verifies the assembly-optimized NTT implementations for Kyber, NTRU, and Saber on Cortex-M4 and Skylake with CryptoLine, and is included in Appendix A.2.

> Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. 2022. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022. Paper. Reference [HLS+22].
>
> **Summary:** We verified the assembly-optimized NTT-based polynomial multiplications in Kyber, Saber, and NTRU on Cortex-M4 and on Skylake with AVX2 with the domain-specific language CryptoLine. I rewrote the assembly implementation of the polynomial multiplication for NTRU on Cortex-M4, explained the program structure of the assembly implementations on Cortex-M4, and drew some figures in the paper.

I rewrote the NTT for NTRU on Cortex-M4 and explained the internal of the Armv7E-M assembly implementations on Cortex-M4.

Revisiting large integer multiplications in RSA. The following published work essentially follows the advancement of NTT implementations in lattice-based cryptosystems, revisits the uses of NTT-based integer multiplications, and is included in Appendix B.1.

> Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number–Theoretic Transforms. In *International Workshop on Security*, pages 3-23. Springer, 2022. Paper. Artifact. IACR ePrint. Reference [BHK+22].
>
> **Summary:** We explored the FFT-based integer multiplications on Cortex-M3 and Cortex-M55, and found that the crossover point of the bit-size/performance of the FFT-based and the $\Theta(n^2)$ non-FFT-based integer multiplications is around 2048 bits. I was involved in the Cortex-M3 implementation.

I reviewed the implementations and ensured the implementations were aligned with the state-of-the-art during the period.

The remaining three published works prior to my PhD studies were included in my master's thesis.

> Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on*

Cryptographic Hardware and Embedded Systems, 2022(1):127-151, 2021. Paper. Artifact. Talk. Slides. IACR ePrint. Reference [ACC+21].

Summary: Continuing the Saber optimizations on Cortex-M4 by [CHK+21], we further optimized Saber on Cortex-M3, the memory footprint on Cortex-M3 and Cortex-M4, and the masked implementation on Cortex-M4, and concluded that NTT is the fastest approach for Saber. I proposed the ideas, implemented the Cortex-M4 optimizations and the fastest approach on Cortex-M3, and wrote the corresponding parts of the paper.

Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):349-371, 2022. Paper. Artifact. Talk. IACR ePrint. Reference [AHY22].

Summary: We revisited NTRU and NTRU Prime on Cortex-M4. We improved the initial butterflies of the Good–Thomas FFT, and the odd-radix butterflies, and proposed incomplete Good–Thomas FFT for code-size optimization. I proposed the optimizations, implemented the optimizations, and wrote the paper.

Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-244, 2021. Paper. IACR ePrint. Reference [BHK+21].

Summary: We rediscovered the multiplicative form of Barrett reduction, discovered a correspondence between Montgomery and Barrett multiplications, implemented the optimizations on Cortex-A72 and Apple M1 with Armv8-A Neon, and integrated the optimizations to Dilithium, Kyber, and Saber. I implemented the optimizations and wrote the corresponding parts of the paper.

1.6 Structure of This Thesis

Part I: Mathematical foundations. Chapter 2 reviews the algebraic background, Chapter 3 goes through the mathematical aspect of modular multiplications, Chapter 4 goes through some fast homomorphisms for polynomial multiplications, Chapter 5 discusses how to adjoin elements to the coefficient rings enabling fast homomorphisms, Chapter 6 analyzes the choices of the polynomial moduli, and Chapter 7 formalizes vectorization.

Part II: General guide for optimizations on target platforms. Chapter 8 reviews the target platforms, Chapter 9 goes through the implementations of modular multiplications on our target platforms, and Chapter 10 goes through a general guide for implementing the transformations.

Part III: Applications to lattice-based cryptosystems. Chapter 13 goes through the implementations of Kyber, Chapter 12 goes through the implementations of Dilithium, Chapter 14 goes through the implementations of NTRU, Chapter 15 goes through the implementations of NTRU Prime, and Chapter 16 goes through the implementations of Saber.

This thesis was written as a monograph on optimizing polynomial multiplications and was based on several publications. The mathematical background sections were merged, the implementation sections of modular arithmetic were merged, and common strategies on optimizing the transformations were also merged. Necessary backgrounds on the basics of algebra and target instruction set architectures and platforms were also reviewed for completeness. As for the applications to lattice-based cryptosystems, the implementation sections were reordered and merged in the platform-wise fashion. The preliminary sections were based on the specifications of the target lattice-based cryptosystems.

Part I

Mathematical Foundations

Chapter 2

Algebraic Background

This chapter reviews the algebraic background for this thesis. We will review the notions of rings, modules, and associative algebras, and refer to [Jac12a, Jac12b] for further reading. Polynomial ring is a classical example of an associative algebra, which is a ring and a module at the same time. From the computational point of view, the ring multiplication is typically implemented with a series of fast transformations, and the module view fits well on mapping the algebraic transformations to actual implementations. Readers familiar with all these algebraic structures can freely skip this chapter. In the rest of this chapter, we assume all the readers are familiar with sets, functions, products of sets, equivalence relations, equivalent classes, and representatives of equivalence classes.

2.1 Rings and Fields

Definition 1 (Monoid)**.** For a set U of elements and a binary function $\xi : U \times U \to U$, we call the pair (U, ξ) a **monoid** if all of the following hold.

$$\begin{cases} \exists e \in U, \forall a \in U, & \xi(a, e) = a = \xi(e, a). \\ \forall a, b, c \in U, & \xi\left(a, \xi(b, c)\right) = \xi\left(\xi(a, b), c\right). \end{cases}$$

For simplicity, we usually write ξ in the infix fashion and denote it as $\cdot$. Frequently, we also denote ab for $a \cdot b$ when the context is clear. Since e is in fact uniquely determined, we call e "the" identity of U and denote it as 1. We also denote a monoid as $(U, \cdot, 1)$. For a subset $V \subset U$ containing the identity 1 of U, we denote $\cdot|_{V \times V}$ as the restriction of $\cdot$ at $V \times V$. If $\mathrm{Img}\left(\cdot|_{V \times V}\right) \subset V$

and $(V, \cdot|_{V\times V}, 1)$ is a monoid, we call $(V, \cdot|_{V\times V}, 1)$ a **submonoid** of $(U, \cdot, 1)$. Img maps a function to its image.

Definition 2 (Group)**.** For a monoid $(U, \cdot, 1)$, we call it a **group** if

$$\forall a \in U, \exists b \in U, ab = 1 = ba.$$

Similarly, for a subset $V \subset U$ containing the identity 1 of U, we call $(V, \cdot|_{V\times V}, 1)$ a **subgroup** of $(U, \cdot, 1)$ if $\operatorname{Img}(\cdot|_{V\times V}) \subset V$ and $(V, \cdot|_{V\times V}, 1)$ is a group. For a group $(G, \cdot, 1)$, we call it an **abelian group** if

$$\forall a, b \in G, ab = ba.$$

For an abelian group, we denote the binary function additively as + and the identity as 0. Conventionally, we denote the binary function of a submonoid as the binary function of the monoid, and the binary function of a subgroup as the binary function of the group.

Definition 3 (Monoid/group homomorphism)**.** Let $(U_0, \cdot_0, 1_0)$ and $(U_1, \cdot_1, 1_1)$ be monoids. For a function $f : U_0 \to U_1$, we call it a **monoid homomorphism** if all of the following hold.

$$\begin{cases} f(1_0) = 1_1. \\ \forall a, b \in U_0, f(ab) = f(a)f(b). \end{cases}$$

If $(U_0, \cdot_0, 1_0)$ and $(U_1, \cdot_1, 1_1)$ are groups, we call f a **group homomorphism**.

For a monoid/group homomorphism f, if f is injective, we call it a **monomorphism**; if f is surjective, we call it an **epimorphism**; and if f is bijective, we call it an **isomorphism**. For a pair (U_0, U_1) of monoids or groups, we call U_0 and U_1 isomorphic if there is an isomorphism between them.

Definition 4 (Ring)**.** For a set R and binary functions $+, \cdot : R \times R \to R$, we call the tuple $(R, +, \cdot, 0, 1)$ a **ring** if $(R, +, 0)$ is a group, $(R, \cdot, 1)$ is a monoid, and

$$\forall a, b, c \in R, (a(b + c) = ab + ac) \land ((b + c)a = ba + ca).$$

For a ring $(R, +, \cdot, 0, 1)$, one can show that $(R, +, 0)$ is in fact an abelian group. When the context is clear, we denote the ring $(R, +, \cdot, 0, 1)$ as R. Conventionally, we call $(R, +, 0)$ the additive group of R and $(R, \cdot, 1)$ the multiplicative monoid of R. Additionally, we call R a **commutative ring** if

$$\forall a, b \in R, ab = ba.$$

For a subset $V \subset R$ containing the identities 0 and 1, we call $(V, +, \cdot, 0, 1)$ a **subring** of R if $(V, +, 0)$ is a subgroup of $(R, +, 0)$ and $(V, \cdot, 1)$ is a submonoid of

$(R, \cdot, 1)$. For a subset $I \subset R$ containing 0, we call it a **left ideal** of R if $(I, +, 0)$ is a subgroup of $(R, +, 0)$ and

$$\forall r \in R, \forall i \in I, ri \in I.$$

Similarly, we call I a **right ideal** if $ir \in I$ and a **two-sided ideal** if $ri, ir \in I$. In this thesis, rings are commutative and there are only two-sided ideals in commutative rings. For simplicity, we use "rings" for commutative rings and "ideals" for two-sided ideals unless specified. For two ideals I and J of R, we call them **coprime ideals** if

$$R = I + J \coloneqq \{i + j | i \in I, j \in J\}.$$

Furthermore, for a finite set of ideals, we call them **pair-wise coprime ideals** if any two of them are coprime. For an ideal I of R, we call it a **principal ideal** if

$$\exists a \in R, I = \{ar | r \in R\}.$$

For a principal ideal I, we denote it as $\langle a \rangle$ if $I = \{ar | r \in R\}$. For an element $r \in R$, we denote the set $\{r + i | i \in I\}$ as $r \bmod I$. Consider the following quotient set

$$R/I \coloneqq \{r \bmod I | r \in R\}.$$

We turn it into a ring by introducing the following ring operations.

$$\begin{cases} + = & (a \bmod I, b \bmod I) \mapsto (a + b) \bmod I. \\ \cdot = & (a \bmod I, b \bmod I) \mapsto ab \bmod I. \end{cases}$$

R/I is called the **quotient ring modulo** I. Finally, all rings contain at least two elements in this thesis.

Definition 5 (Ring homomorphism)**.** For rings R_0 and R_1 and a function $f : R_0 \to R_1$, we call f a **ring homomorphism** if it is a monoid homomorphism between the multiplicative monoids and a group homomorphism between the additive groups.

Similar to monoid and group homomorphisms, we call a ring homomorphism monomorphism if it is injective, epimorphism if it is surjective, and isomorphism if it is bijective. For pair-wise coprime ideals $I_0, \dots, I_{d-1}$ of a (possibly non-commutative) ring R with the intersection $I = \bigcap_{i=0}^{d-1} I_i$, we have the following isomorphism:

$$R/I \cong \prod_{i=0}^{d-1} R/I_i \,.$$

This is the Chinese remainder theorem (CRT) for (possibly non-commutative) rings. For a sequence of elements $e_0, \ldots, e_{d-1}$, we call them **pair-wise orthogonal idempotent elements** if $e_i e_j = \delta_{i,j}$ for all i, j where $\delta_{i,j}$ is the Kronecker's delta function. We identify the unique sequence of pair-wise orthogonal idempotent elements $e_0, \ldots, e_{d-1} \in R/I$ and find $(a_0, \ldots, a_{d-1}) \mapsto \sum_{i=0}^{d-1} e_i a_i : \prod_{i=0}^{d-1} R/I_i \to R/I$ the inverse of $a \mapsto (a \bmod I_0, \ldots, a \bmod I_{d-1}) : R/I \to \prod_{i=0}^{d-1} R/I_i$. This follows from $R/I_i \cong (R/I)/(I_i/I)$ and [Bou89, Proposition 10, Section 8.11, Chapter I]. When R is commutative, we have

$$R \Big/ \prod_{i=0}^{d-1} I_i \cong \prod_{i=0}^{d-1} R/I_i \, .$$

If R is commutative and $\langle a_0 \rangle, \ldots, \langle a_{d-1} \rangle$ are all principal ideals of R, we have

$$R \Big/ \left\langle \prod_{i=0}^{d-1} a_i \right\rangle \cong \prod_{i=0}^{d-1} R/\langle a_i \rangle \, .$$

Definition 6 (Integral domain/principal idea domain)**.** For a commutative ring R, we call it an **integral domain** if

$$\forall a, b \in R, ab = 0 \longrightarrow (a = 0 \vee b = 0) \, .$$

If all the ideals of an integral domain are principal, we call the integral domain a **principal ideal domain**.

Definition 7 (Field)**.** For a ring $(F, +, \cdot, 0, 1)$, we call it a **field** if $(F - \{0\}, \cdot, 1)$ is a group.

Fields are necessarily principal ideal domains. One can show that if $|F| < \infty$, then fields and integral domains coincide [Jac12a, Excercise 1, Section 2.2].

Examples. Below we review some classical examples. The set of integers $\mathbb{Z}$ and the multiplication $\cdot$ form a monoid $(\mathbb{Z}, \cdot, 1)$. If we replace multiplication by addition, we have an abelian group $(\mathbb{Z}, +, 0)$. It is clear that $(\mathbb{Z}, +, \cdot, 0, 1)$ is a ring and also an integral domain. For a positive integer q, the set $q\mathbb{Z} \coloneqq \{qz | z \in \mathbb{Z}\}$ is a principal ideal of $\mathbb{Z}$. Since all the ideals of $\mathbb{Z}$ are principal, $\mathbb{Z}$ is a principal ideal domain. The quotient ring $\mathbb{Z}/q\mathbb{Z}$ is often denoted as $\mathbb{Z}_q$[1]. For the underlying group structure $(\mathbb{Z}_q, +, 0)$, one can show that for an integer g coprime to q,

---

[1] One should note that $\mathbb{Z}_q$ also stands for p-adic integers when $q = p$ is a prime. This thesis does not consider p-adic integers and deliberately uses $\mathbb{Z}_q$ with a possibly composite integer q for the quotient ring $\mathbb{Z}/q\mathbb{Z}$.

$z \mapsto zg : \mathbb{Z}_q \to \mathbb{Z}_q$ is a group automorphism. $(\mathbb{Z}_q, +, \cdot, 0, 1)$ is a field if and only if q is a prime. When q is a composite number with coprime factorization $q = q_0 q_1$, $\mathbb{Z}_q \cong \mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1}$ is a ring homomorphism. Since fields with q elements are isomorphic, we uniquely identify them as $\mathbb{F}_q$. For a finite field $\mathbb{F}_q$, q is a power of a prime. Fields are not restricted to finite ones. For the set $\mathbb{Q}$ of rational numbers and the set $\mathbb{R}$ of real numbers, $(\mathbb{Q}, +, \cdot, 0, 1)$ and $(\mathbb{R}, +, \cdot, 0, 1)$ are also field with $+$ and $\cdot$ the corresponding rational number and real number additions and multiplications.

2.2 Modules and Associative Algebras

Definition 8 (Module)**.** For an abelian group M, a ring R, and a function $\cdot : R \times M \to M$, we call the tuple $(M, \cdot)$ a **left R-module** if all of the following hold.

$$\begin{cases} \forall r \in R, \forall \boldsymbol{a}, \boldsymbol{b} \in M, & r \cdot (\boldsymbol{a} + \boldsymbol{b}) = r \cdot \boldsymbol{a} + r \cdot \boldsymbol{b}. \\ \forall r, s \in R, \forall \boldsymbol{a} \in M, & ((r+s) \cdot \boldsymbol{a} = r \cdot \boldsymbol{a} + s \cdot \boldsymbol{a}) \wedge ((rs) \cdot \boldsymbol{a} = r \cdot (s \cdot \boldsymbol{a})). \\ \forall \boldsymbol{a} \in M, & 1 \cdot \boldsymbol{a} = \boldsymbol{a}. \end{cases}$$

For a left R-module $(M, \cdot)$, we call $\cdot$ the scalar multiplication and omit it when the context is clear. If $\cdot$ has the signature $M \times R \to M$, we call $(M, \cdot)$ a **right R-module** if the conditions hold with scalars multiplied from the right. Since R is commutative in this thesis, studying left R-modules is equivalent to studying right R-modules and we omit the distinction between left and right modules. Let M be an R-module. For a set of elements $B \subset M$, we call B linearly independent if for an arbitrary positive integer $n \leq |B|$,

$$\forall r_0, \ldots, r_{n-1} \in R, \forall \{\gamma_0, \ldots, \gamma_{n-1}\} \subset B, \sum_{i=0}^{n-1} r_i \gamma_i = 0 \longrightarrow r_0 = \cdots = r_{n-1} = 0.$$

For a linearly independent set $B \subset M$, we call it a **basis of M** if

$$\forall a \in M, \exists n \in \mathbb{Z}_{>0}, \exists r_0, \ldots, r_{n-1} \in R, \exists \{\gamma_0, \ldots, \gamma_{n-1}\} \subset B, a = \sum_{i=0}^{n-1} r_i \gamma_i.$$

A **free module** is a module with a basis. In this thesis, all modules are free modules. The cardinality of a basis of a free module is uniquely determined in our context and is called the rank of a module.

Definition 9 (Module homomorphism and dual module)**.** Let M and N be two R-modules and $f : M \to N$ be a function. We call f a **module homomorphism** if all of the following hold.

$$\begin{cases} \forall \boldsymbol{a}, \boldsymbol{b} \in M, & f(\boldsymbol{a}+\boldsymbol{b}) = f(\boldsymbol{a}) + f(\boldsymbol{b}). \\ \forall r \in R, \forall \boldsymbol{a} \in M, & f(r\boldsymbol{a}) = rf(\boldsymbol{a}). \end{cases}$$

We denote the set of module homomorphisms from M to N as $\mathrm{Hom}(M, N)$. When $N = R$, we call $\mathrm{Hom}(M, R)$ the **dual of** M and denote it as M^*.

For an R-module M, if M has finite rank and R is commutative, M^* is an R-module isomorphic to M. For a module M with finite rank and an element $\boldsymbol{a} \in M$, we denote $\boldsymbol{a}^* \in M^*$ for its image under the duality map implementing the isomorphism $M \cong M^*$. For modules M and N of finite ranks and a module homomorphism $f : M \to N$, we define **the dual of** f as the following module homomorphism.

$$f^* : \begin{cases} N^* & \to M^*, \\ \boldsymbol{a}^* & \mapsto \boldsymbol{a}^* \circ f. \end{cases}$$

Definition 10 (Tensor product of modules)**.** Let M and N be R-modules. We define the **tensor product of modules** M **and** N as $F(M \times N)/\sim$ where

- $M \times N$ is the Cartesian product of M and N,
- $F(M \times N)$ is the set of all formal linear combinations of elements from $M \times N$ over R, and
- $\sim$ is the equivalence relation generated by

$$\forall \boldsymbol{a}, \boldsymbol{b} \in M, \forall \boldsymbol{c}, \boldsymbol{d} \in N, (\boldsymbol{a}+\boldsymbol{b}, \boldsymbol{c}+\boldsymbol{d}) \sim (\boldsymbol{a}, \boldsymbol{c}) + (\boldsymbol{a}, \boldsymbol{d}) + (\boldsymbol{b}, \boldsymbol{c}) + (\boldsymbol{b}, \boldsymbol{d}).$$
$$\forall r \in R, \forall \boldsymbol{a} \in M, \forall \boldsymbol{b} \in N, (r\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a}, r\boldsymbol{b}).$$

We denote an equivalence class in $F(M \times N)/\sim$ as $\boldsymbol{a} \otimes \boldsymbol{b}$ and $F(M \times N)/\sim$ as $M \otimes N$, and also regard $M \otimes N$ as an R-module by defining the scalar multiplication as

$$\forall r \in R, \forall \boldsymbol{a} \in M, \forall \boldsymbol{b} \in N, r(\boldsymbol{a} \otimes \boldsymbol{b}) = (r\boldsymbol{a}) \otimes \boldsymbol{b}.$$

Definition 11 (Tensor product of module homomorphisms)**.** For module homomorphisms $f_0 : M_0 \to N_0$ and $f_1 : M_1 \to N_1$, we define the **tensor product of** f_0 **and** f_1 as follows.

$$f_0 \otimes f_1 : \begin{cases} M_0 \otimes M_1 & \to N_0 \otimes N_1. \\ \boldsymbol{a} \otimes \boldsymbol{b} & \mapsto f_0(\boldsymbol{a}) \otimes f_1(\boldsymbol{b}). \end{cases}$$

One can show that $f_0 \otimes f_1$ is a module homomorphism.

Definition 12 (Associative algebra)**.** Let $\mathcal{A}$ be an R-module and $\cdot : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ be a binary function. If $(\mathcal{A}, +, \cdot)$ becomes a ring where $+$ comes from the abelian group structure of $\mathcal{A}$ and

$$\forall r \in R, \forall \boldsymbol{a}, \boldsymbol{b} \in \mathcal{A}, r(\boldsymbol{ab}) = (r\boldsymbol{a})\boldsymbol{b} = \boldsymbol{a}(r\boldsymbol{b}),$$

we call $\mathcal{A}$ an **associative R-algebra**. Conventionally, we call it an R-algebra or algebra when the context is clear.

Definition 13 (Algebra homomorphism)**.** For algebras $\mathcal{A}$ and $\mathcal{B}$ and a module homomorphism $f : \mathcal{A} \to \mathcal{B}$, if f is also a ring homomorphism, we call it an **algebra homomorphism**.

Definition 14 (Tensor product of algebras)**.** For algebras $\mathcal{A}$ and $\mathcal{B}$, we naturally have the tensor product $\mathcal{A} \otimes \mathcal{B}$ of the underlying modules. Consider a binary function $\cdot : (\mathcal{A} \otimes \mathcal{B}) \times (\mathcal{A} \otimes \mathcal{B}) \to (\mathcal{A} \otimes \mathcal{B})$ defined by

$$\forall \boldsymbol{a} \otimes \boldsymbol{b}, \boldsymbol{c} \otimes \boldsymbol{d} \in \mathcal{A} \otimes \mathcal{B}, (\boldsymbol{a} \otimes \boldsymbol{b}) \cdot (\boldsymbol{c} \otimes \boldsymbol{d}) = (\boldsymbol{ac}) \otimes (\boldsymbol{bd}),$$

we have $(\mathcal{A} \otimes \mathcal{B}, \cdot)$ as an algebra. We call $\mathcal{A} \otimes \mathcal{B}$ the **tensor product of algebras $\mathcal{A}$ and $\mathcal{B}$**.

Definition 15 (Tensor product of algebra homomorphisms)**.** For algebra homomorphisms $f_0 : \mathcal{A}_0 \to \mathcal{B}_0$ and $f_1 : \mathcal{A}_1 \to \mathcal{B}_1$, the module homomorphism $f_0 \otimes f_1$ is also an algebra homomorphism and we call it a **tensor product of algebra homomorphisms**.

Example 1 ($\mathbb{Z}$-modules)**.** For an element g in the abelian group G and an integer n, we can naturally define $ng = \underbrace{g + \cdots + g}_{n}$. Therefore, G is a $\mathbb{Z}$-module.

Example 2 (Module R^n)**.** Let n be a positive integer and R^n be the n-fold product of R. R^n is naturally an abelian group and we turn it into an R-module by defining the scalar multiplication as $r \cdot (s_0, \ldots, s_{n-1}) = (rs_0, \ldots, rs_{n-1})$. Clearly, $\{e_0, \ldots, e_{n-1}\}$ is a basis of R^n where e_i stands for the tuple with 1 at the ith position and 0 elsewhere.

Example 3 ($\mathbb{Z}$-algebras)**.** Let R be a (possibly non-commutative) ring. Clearly, R is a $\mathbb{Z}$-module. Since we also have

$$\forall n \in \mathbb{Z}, \forall \boldsymbol{a}, \boldsymbol{b} \in R, n(\boldsymbol{ab}) = (n\boldsymbol{a})\boldsymbol{b} = \boldsymbol{a}(n\boldsymbol{b}),$$

R is in fact a $\mathbb{Z}$-algebra.

Example 4 (Polynomial ring $R[x]$)**.** Let R be a ring. We define a polynomial over R as a sequence of elements drawn from R where all but finitely many entries are non-zeros. For an indeterminate x, we denote a polynomial $\boldsymbol{a}$ as $\sum_{i=0}^{n-1} a_i x^i$ for a positive integer n and elements $a_i \in R$. Conventionally, if any of a_i is non-zero, we define the degree $\deg(\boldsymbol{a})$ of $\boldsymbol{a}$ as the largest integer i with $a_i \neq 0$ and the size of $\boldsymbol{a}$ as $\deg(\boldsymbol{a}) + 1$. If $\boldsymbol{a}$ is zero everywhere, its degree is defined as $-\infty$[2]. For a non-zero polynomial $\boldsymbol{a}$, we call it a **monic polynomial** if $a_{\deg(\boldsymbol{a})} = 1$. The set of polynomials over R is denoted as $R[x]$ and is in fact a ring. We define the ring operations as follows.

- Addition:

$$\forall \sum_{i=0}^{n-1} a_i x^i, \sum_{i=0}^{n-1} b_i x^i \in R[x], \sum_{i=0}^{n-1} a_i x^i + \sum_{i=0}^{n-1} b_i x^i = \sum_{i=0}^{n-1} (a_i + b_i) x^i.$$

- Multiplication:

$$\begin{aligned} &\forall \sum_{i=0}^{n-1} a_i x^i, \sum_{i=0}^{m-1} b_i x^i \in R[x], \\ &\left(\sum_{i=0}^{n-1} a_i x^i \right) \left(\sum_{i=0}^{m-1} b_i x^i \right) = \sum_{i=0}^{n+m-2} \left(\sum_{h+k=i} a_h b_k \right) x^i. \end{aligned}$$

One can verify $R[x]$ is a ring with the additive identity 0 and the multiplicative identity 1. The ring multiplication also defines the scalar multiplication as an R-module, and hence $R[x]$ is in fact an algebra.

Example 5 (Polynomial ring $\mathbb{F}[x]$)**.** For a polynomial ring $\mathbb{F}[x]$ with $\mathbb{F}$ a field, one can show that $\mathbb{F}[x]$ is a principal ideal domain.

Example 6 (Polynomial ring $R[x]/\langle \boldsymbol{g} \rangle$ for a non-zero polynomial $\boldsymbol{g}$)**.** For a polynomial ring $R[x]$ and a non-zero polynomial $\boldsymbol{g} \in R[x]$, we have the principal ideal $\langle \boldsymbol{g} \rangle = \boldsymbol{g}R$ and the quotient ring $R[x]/\langle \boldsymbol{g} \rangle$. The R-module structure also holds in $R[x]/\langle \boldsymbol{g} \rangle$ so $R[x]/\langle \boldsymbol{g} \rangle$ is an algebra. If $\boldsymbol{g} = x^n$ for a positive integer n, we denote $R[x]/\langle x^n \rangle$ as $R[x]_{<n}$.

Example 7 (Polynomial ring $\mathbb{Z}[x]/\langle \Phi_n(x) \rangle$ for the nth cyclotomic polynomial $\Phi_n(x)$)**.** For a polynomial ring $R[x]$, a polynomial is a **irreducible polynomial** if it cannot be factored into two non-constant polynomials. For a positive integer n, **the nth cyclotomic polynomial** $\Phi_n(x)$ is the unique irreducible polynomial that is a factor of $x^n - 1$ and not a factor of $x^m - 1$ for any positive integer $m < n$. One can show that $\mathbb{Q}[x]/\langle \Phi_n(x) \rangle$ and $\mathbb{R}[x]/\langle \Phi_n(x) \rangle$ are fields.

---

[2] In mathematics, the degree of 0 is either undefined, or defined as $0, -1, -\infty$. We define its degree as $-\infty$ since we are multiplying polynomials and $\deg(0) = -\infty$ is compatible with multiplying a non-zero polynomial by a zero.

Example 8 (The set $\mathbb{C}$ of complex numbers)**.** Consider the field $\mathbb{R}[x]/\langle x^2+1\rangle$ where $x^2+1=\Phi_4(x)$. The elements in $\mathbb{R}[x]/\langle x^2+1\rangle$ are called complex numbers and we call $\mathbb{C}=\mathbb{R}[x]/\langle x^2+1\rangle$ the set of complex numbers. For the nth cyclotomic polynomial $\Phi_n(x)$, it factors into $\prod_{i\perp n}\left(x-e^{\frac{2i\pi\sqrt{-1}}{n}}\right)$ where $e=\sum_{n=0}^{\infty}\frac{1}{n!}$ and $\sqrt{-1}\in\mathbb{R}[x]/\langle x^2+1\rangle$ is a root of $x^2+1=0$.

Example 9 (Group algebras)**.** Let G be a group and R be a ring. The **group algebra** $R[G]$ is defined as the set of formal linear combinations of elements in G with coefficients in R where all but finitely many coefficients are non-zeros. Concretely, $R[G]$ consists of all the elements of the form

$$\sum_{g\in G}a_g g$$

with $|\{a_g|g\in G\}|<\infty$. Clearly, $R[G]$ is an algebra and G is a basis of $R[G]$. For a positive integer n, we have $R[x]/\langle x^n-1\rangle\cong R\left[\mathbb{Z}_n\right]$ with the map induced by $\forall i\in\mathbb{Z}_n, x^i\mapsto i$.

Example 10 (Tensor products of group algebras)**.** Let G, G_0, G_1 be groups with $G\cong G_0\times G_1$ and R be a ring. We have group algebras $R[G], R[G_0], R[G_1]$. Since $G\cong G_0\times G_1$, we can write all the elements in $R[G]$ as formal linear combinations of the form

$$\sum_{g_0\in G_0}\sum_{g_1\in G_1}a_{g_0,g_1}(g_0,g_1).$$

One can show that $(g_0,g_1)\mapsto g_0\otimes g_1$ induces an algebra isomorphism from $R\left[G_0\times G_1\right]$ to $R\left[G_0\right]\otimes R\left[G_1\right]$. For coprime integers q_0, q_1, we know that $\mathbb{Z}_{q_0q_1}\cong\mathbb{Z}_{q_0}\times\mathbb{Z}_{q_1}$ as groups. Therefore, we have algebra isomorphisms $R\left[\mathbb{Z}_{q_0q_1}\right]\cong R\left[\mathbb{Z}_{q_0}\right]\otimes R\left[\mathbb{Z}_{q_1}\right]$ and $R[x]/\langle x^{q_0q_1}-1\rangle\cong R[y]/\langle y^{q_0}-1\rangle\otimes R[z]/\langle z^{q_1}-1\rangle$.

Chapter 3

Modular Multiplications

This chapter goes through several modular multiplications. For a modulus q, and two integers a, b, we want to compute an integer c with $c \equiv ab \pmod{q}$ with additions, subtractions, multiplications, and logical operations only. We denote $\mathtt{R}$, typically a power of two, the number of integers representable by a machine word. In the lattice-based cryptosystems covered by this thesis, a, b, c are integers fit into machine words. This implies very fast division by $\mathtt{R}$ with flooring and reduction modulo $\mathtt{R}$.

3.1 Numbers

We denote $\mathbb{Z}$ as the set of integers, $\frac{1}{2} + \mathbb{Z} = \left\{z + \frac{1}{2} | z \in \mathbb{Z}\right\}$ as the set of half integers, $\mathbb{Q}$ as the set of rational numbers, $\mathbb{R}$ as the set of real numbers, and $\mathbb{C}$ as the set of complex numbers. For an integer n, we define $\mathbb{Z}_{<n}$ as the set of integers smaller than n, $\mathbb{Z}_{\leq n} \coloneqq \mathbb{Z}_{<n} \cup \{n\}$, $\mathbb{Z}_{>n}$ as the set of integers greater than n, and $\mathbb{Z}_{\geq n} \coloneqq \mathbb{Z}_{>n} \cup \{n\}$. If n is a power of two, we call $\log_2 n$ the precision. For real numbers $l \leq r$, we define $[l, r] \coloneqq \{q \in \mathbb{R} | l \leq q \wedge q \leq r\}$ and $[l, r) \coloneqq \{q \in \mathbb{R} | l \leq q \wedge q < r\}$.

Definition 16 (Unsigned multi-word representation)**.** Let $\mathtt{R}$ be a power of two and a a positive integer. The unsigned multi-word representation of a is defined as the unique tuple $\{b_i\} \subset [0, \mathtt{R}) \cap \mathbb{Z}$ such that

$$\sum_i b_i \mathtt{R}^i = a.$$

Furthermore, we denote $(b_i) = \mathrm{usplit}_{\log_2 \mathtt{R}}(a)$.

Definition 17 (Signed multi-word representation)**.** Let $\mathtt{R}$ be a power of two and a an integer. The signed multi-word representation of a is defined as the unique tuple $\{b_i\} \subset \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$ such that

$$\sum_i b_i \mathtt{R}^i = a.$$

Furthermore, we denote $(b_i) = \mathrm{ssplit}_{\log_2 \mathtt{R}}(a)$.

If $b_i = 0$ for all $i \geq 2$, we denote b_0 as $\mathtt{ulo}_{\log_2 \mathtt{R}}(a)$ and b_1 as $\mathtt{uhi}_{\log_2 \mathtt{R}}(a)$ in the unsigned case, and b_0 as $\mathtt{slo}_{\log_2 \mathtt{R}}(a)$ and b_1 as $\mathtt{shi}_{\log_2 \mathtt{R}}(a)$ in the signed case.

3.2 Integer Approximations and Modular Reductions

Conventionally, $\mathbb{Z}_q$ is identified as a set of q consecutive integers. Popular choices are $[0, q) \cap \mathbb{Z}$ and $\left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$. By default, we identify $\mathbb{Z}_q$ as the set $\left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$.

Definition 18 (Integer approximation [HKS23])**.** For a real number $\delta > 0$ and a function $\llbracket\rrbracket : \mathbb{R} \to \mathbb{Z}$, we call $\llbracket\rrbracket$ a δ-integer-approximation if

$$\forall r \in \mathbb{R}, |\llbracket r \rrbracket - r| \leq \delta.$$

We call $\llbracket\rrbracket$ an integer approximation as long as there is a δ such that $\llbracket\rrbracket$ is a δ-integer-approximation. For an integer approximation and a positive integer greater than one, they naturally define a "modular reduction."

Definition 19 (Modular reduction [HKS23])**.** For an integer approximation $\llbracket\rrbracket$ and an integer $q > 1$, we define the corresponding modular reduction $\bmod^{\llbracket\rrbracket} q : \mathbb{Z} \to \mathbb{Z}$ as:

$$\forall z \in \mathbb{Z}, z \bmod^{\llbracket\rrbracket} q := z - \left\llbracket \frac{z}{q} \right\rrbracket q$$

and $\left| \bmod^{\llbracket\rrbracket} q \right| := \max_{z \in \mathbb{Z}} \left| z \bmod^{\llbracket\rrbracket} q \right|$.

Corollary 1. For an integer $q > 1$ and integers a, b, if $a \equiv b \pmod{q}$, then for an arbitrary integer approximation $\llbracket\rrbracket$, we have $a \bmod^{\llbracket\rrbracket} q \equiv b \bmod^{\llbracket\rrbracket} q \pmod{q}$.

Corollary 2. For an integer approximation $\llbracket\rrbracket$ and an integer $q > 1$, we have $\left| \mathrm{Img}\left(\bmod^{\llbracket\rrbracket} q \right) \right| \geq q$.

Lemma 1. For an integer approximation $\llbracket\rrbracket$ and an integer $q > 1$, we have

$$\forall z \in \mathbb{Z}, \begin{cases} \left\llbracket \frac{z}{q} \right\rrbracket q = z - z \bmod^{\llbracket\rrbracket} q, \\ z \equiv z \bmod^{\llbracket\rrbracket} q \pmod{q}. \end{cases}$$

Corollary 3. Let q and $\mathtt{R}$ be coprime integers greater than 1, $\llbracket\rrbracket_0$ and $\llbracket\rrbracket_1$ be integer approximations, and $\bmod^{\llbracket\rrbracket_0} q$ and $\bmod^{\llbracket\rrbracket_1} \mathtt{R}$ be the corresponding modular reductions. If $\left|\mathrm{Img}\left(\bmod^{\llbracket\rrbracket_1}\mathtt{R}\right)\right| = \mathtt{R}$, then for an integer z, we have

$$\left\llbracket \frac{z\mathtt{R}}{q} \right\rrbracket_0 \bmod^{\llbracket\rrbracket_1}\mathtt{R} = \left(z\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right)\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1}\mathtt{R}.$$

Proof.

$$\begin{aligned} \left\llbracket \frac{z\mathtt{R}}{q} \right\rrbracket_0 \bmod^{\llbracket\rrbracket_1}\mathtt{R} &= \frac{z\mathtt{R} - \left(z\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right)}{q} \bmod^{\llbracket\rrbracket_1}\mathtt{R} \\ &= \left(z\mathtt{R} \bmod^{\llbracket\rrbracket_0} q\right)\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1}\mathtt{R}. \end{aligned}$$

□

Relations to equivalence classes. We define the floor function $\lfloor\rfloor$, the ceiling function $\lceil\rceil$, and the rounding-half-up function $\lfloor\rceil$ as follows.

$$\lfloor\rfloor : \begin{cases} \mathbb{R} \to & \mathbb{Z}, \\ r \mapsto & \max\{z \in \mathbb{Z} | z \leq r\}. \end{cases}$$

$$\lfloor\rceil : \begin{cases} \mathbb{R} \to & \mathbb{Z}, \\ r \mapsto & \lfloor r + 0.5 \rfloor . \end{cases}$$

$$\lceil\rceil : \begin{cases} \mathbb{R} \to & \mathbb{Z}, \\ r \mapsto & -\lfloor -r \rfloor . \end{cases}$$

See Figure 3.1 for an illustration of $\lfloor\rfloor$, Figure 3.2 for $\lfloor\rceil$, and Figure 3.3 for $\lceil\rceil$. Obviously, all are 1-integer-approximations. We have $\mathrm{Img}\left(\bmod^{\lfloor\rceil} q\right) = \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ and $\mathrm{Img}\left(\bmod^{\lfloor\rfloor} q\right) = [0, q) \cap \mathbb{Z}$.

Figure 3.1: The floor function $\lfloor\rfloor$.

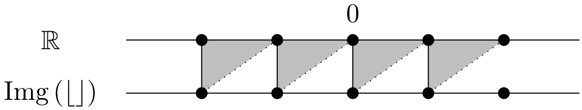


Figure 3.2: The rounding-half-up function $\lfloor\rceil$.

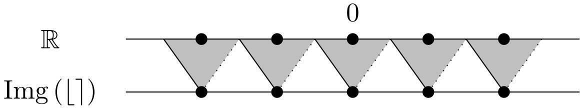
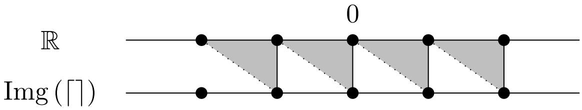


Figure 3.3: The ceiling function $\lceil\rceil$.



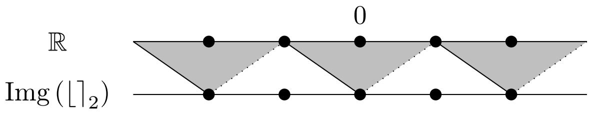
Definition 20 (Quality of modular reduction)**.** For an integer approximation $\llbracket\rrbracket$ and an integer $q > 1$, we define $\frac{\left|\bmod^{\llbracket\rrbracket} q\right|}{\lfloor q/2 \rfloor}$ as the quality of $\bmod^{\llbracket\rrbracket} q$ as a signed modular reduction and $\frac{\left|\bmod^{\llbracket\rrbracket} q\right|}{q-1}$ as the quality of $\bmod^{\llbracket\rrbracket} q$ as an unsigned modular reduction.

Essentially, the quality of a modular reduction measures how far the image is from $\left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ in the signed case and $[0, q) \cap \mathbb{Z}$ in the unsigned case. If we relax the quality of a modular reduction, we have a less accurate integer approximation. We illustrate such an example. Consider the function $\lfloor\rceil_2 = r \mapsto 2\left\lfloor\frac{r}{2}\right\rceil$ mapping a real number to the closest even integer with rounding-up as the tie-breaking rule. See Figure 3.4 for an illustration. Clearly, $\bmod^{\lfloor\rceil_2} q = \bmod^{\lfloor\rceil} 2q$ and the quality of $\bmod^{\lfloor\rceil_2} q$ as a signed modular reduction is 2. The relaxation of the quality of a modular reduction enables highly adaptive solutions for the following obstacles encountered in practice.

- On some platforms, there might be no instructions implementing $\lfloor\rceil$ for signed modular reductions. We instead look into instructions implementing other 1-integer-approximations.

- On some platforms, there might be no instructions implementing 1-integer-approximations at all. In this case, we have to implement an integer approximation with other basic operations and trade the quality of the resulting modular reduction with the efficiency of the integer approximation.

Figure 3.4: Rounding-to-the-nearest-even.



3.3 Montgomery Multiplication

Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers $a, b \in \mathbb{Z}_{\mathtt{R}}$, Montgomery multiplication [Mon85, Sei18] computes a representative of $ab \bmod^{\pm} q$ with a potential scaling. Observe that $ab + \left(ab\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q$ is equivalent to 0 modulo $\mathtt{R}$ and ab modulo q, we have

$$\frac{ab + \left(ab\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \equiv ab\mathtt{R}^{-1} \pmod{q}.$$

To see why this is a reduction, we bound the range as follows:

$$\left|\frac{ab + \left(ab\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}\right| \leq \frac{|ab| + \left| \bmod^{\pm} \mathtt{R}\right| q}{\mathtt{R}}.$$

We can generalize $\bmod^{\pm}$ to arbitrary integer approximations.

Definition 21 (Montgomery multiplication [Mon85, Sei18, HKS23]). Let q and $\mathtt{R}$ be coprime integers greater than 1, $[\![]\!]_0$ and $[\![]\!]_1$ be integer approximations, and $\bmod^{[\![]\!]_0} q$ and $\bmod^{[\![]\!]_1} \mathtt{R}$ be the corresponding modular reductions. For integers a and b, Montgomery multiplication computes a representative of $ab\mathtt{R}^{-1} \bmod q$ as

$$\frac{ab + \left(ab\left(-q^{-1}\right) \bmod^{[\![]\!]_1} \mathtt{R}\right) q}{\mathtt{R}}$$

with the following bound

$$\left|\frac{ab + \left(ab\left(-q^{-1}\right) \bmod^{[\![]\!]_1} \mathtt{R}\right) q}{\mathtt{R}}\right| \leq \frac{|ab| + \left| \bmod^{[\![]\!]_1} \mathtt{R}\right| q}{\mathtt{R}}.$$

There are many ways to mitigate the scaling. One can apply a Montgomery multiplication with the constant $\mathtt{R}^2 \bmod^{\pm} q$ to one of the operands a, b or the result $\frac{ab + \left(ab\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}$. Suppose b is known, we precompute $b\mathtt{R} \bmod^{\pm} q$ and compute the following instead

$$\frac{a\left(b\mathtt{R} \bmod^{\pm} q\right) + \left(a\left(b\mathtt{R} \bmod^{\pm} q\right)\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \equiv ab \pmod{q}.$$

Similarly, the mitigation of scaling also generalizes to arbitrary integer approximations.

Definition 22 (Montgomery multiplication with precomputation [Mon85, Sei18, HKS23])**.** Let q and $\mathtt{R}$ be coprime integers greater than 1, $\llbracket\rrbracket_0$ and $\llbracket\rrbracket_1$ be integer approximations, and $\bmod^{\llbracket\rrbracket_0} q$ and $\bmod^{\llbracket\rrbracket_1} \mathtt{R}$ be the corresponding modular reductions. For integers a, b, and $b' = b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q$ where b' is precomputed, Montgomery multiplication computes a representative of $ab \bmod q$ as

$$\frac{ab' + \left(ab' \left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}},$$

and find the following bound

$$\left| \frac{ab' + \left(ab' \left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} \right| \leq \frac{|a| \left| \bmod^{\llbracket\rrbracket_0} q \right| + \left| \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right| q}{\mathtt{R}}.$$

Corollary 4. Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers a and b, we have

$$\left| \frac{a \left(b\mathtt{R} \bmod^{\pm} q\right) + \left(a \left(b\mathtt{R} \bmod^{\pm} q\right) \left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \right| \leq \frac{q}{2} \left(1 + \frac{|a|}{\mathtt{R}}\right).$$

Furthermore, if $\left|a \left(b\mathtt{R} \bmod^{\pm} q\right)\right| < \frac{\mathtt{R}}{2}$, then

$$\frac{a \left(b\mathtt{R} \bmod^{\pm} q\right) + \left(a \left(b\mathtt{R} \bmod^{\pm} q\right) \left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} = ab \bmod^{\pm} q.$$

In practice, for q an odd integer, $\mathtt{R}$ is chosen as a power of two for efficient computation.

Historical review. [Mon85] proposed the unsigned Montgomery multiplication, and [Sei18] proposed the signed variant along with the following subtractive variant

$$\frac{ab - \left(abq^{-1} \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}.$$

Since $\left(ab \bmod^{\pm} \mathtt{R}\right) = \left(\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right) q \bmod^{\pm} \mathtt{R}\right)$, we can discard the results of the lower parts of the products. This does not hold in Definitions 21 and 22 as $\left(ab \bmod^{\pm} \mathtt{R}\right) = \mathtt{R} - \left(\left(ab \left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) q \bmod^{\pm} \mathtt{R}\right)$ or 0 as integers. The subtractive variant can be implemented as

$$\left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(abq^{-1} \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \right\rfloor.$$

3.4 Barrett Multiplication

Let $q > 1$ be an integer. For two integers a and b, we have $ab \bmod^{\pm} q = ab - \left\lfloor \frac{ab}{q} \right\rceil q$. Barrett multiplication replaces $\left\lfloor \frac{ab}{q} \right\rceil$ with a δ-integer approximation admitting an efficient computation for a small $\delta > 0$.

Definition 23 (Barrett multiplication [Sho, BHK+21, HKS23])**.** Let q and $\mathtt{R}$ be integers greater than 1, $\llbracket\rrbracket_0$ and $\llbracket\rrbracket_1$ be integer approximations, and $\bmod^{\llbracket\rrbracket_0} q$ and $\bmod^{\llbracket\rrbracket_1} \mathtt{R}$ be the corresponding modular reductions. For integers a and b, Barrett multiplication computes a representative of $ab \bmod q$ as

$$ab - \left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q.$$

Theorem 1 (Barrett–Montgomery correspondence [BHK+21, HKS23])**.** Let q and $\mathtt{R}$ be coprime integers greater than 1, $\llbracket\rrbracket_0$ and $\llbracket\rrbracket_1$ be integer approximations, and $\bmod^{\llbracket\rrbracket_0} q$ and $\bmod^{\llbracket\rrbracket_1} \mathtt{R}$ be the corresponding modular reductions. If $\left| \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right| = \mathtt{R}$, then for integers a and b, we have

$$ab - \left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q = \frac{ab' + \left(ab' \left(-q^{-1} \right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right) q}{\mathtt{R}}$$

where $b' = b\mathtt{R} \bmod \llbracket\rrbracket_0\, q$.

Proof.

$$\begin{aligned} ab - \left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q = & \quad ab - \frac{a \llbracket b\mathtt{R}/q \rrbracket_0 - \left(a \llbracket b\mathtt{R}/q \rrbracket_0 \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right)}{\mathtt{R}} q \\ = & \quad \frac{a \left(b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q \right) + \left(a \llbracket b\mathtt{R}/q \rrbracket_0 \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right) q}{\mathtt{R}} \\ = & \quad \frac{ab' + \left(ab' \left(-q^{-1} \right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \right) q}{\mathtt{R}}. \end{aligned}$$

□

Once we determine the integer approximations, Barrett multiplication computes exactly the same result as a specific Montgomery multiplication built upon the corresponding modular reductions. Furthermore, for an integer $\mathtt{R}'$ and an integer approximation $\llbracket\rrbracket$ with $\left| \mathrm{Img} \left(\bmod^{\llbracket\rrbracket} \mathtt{R}' \right) \right| = \mathtt{R}'$, if we have $ab - \left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \in \mathrm{Img} \left(\bmod^{\llbracket\rrbracket} \mathtt{R}' \right)$, we can rewrite the left-hand side as

$$ab - \left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q = \left(\left(ab \bmod^{\llbracket\rrbracket} \mathtt{R}' \right) - \left(\left\llbracket \frac{a \llbracket b\mathtt{R}/q \rrbracket_0}{\mathtt{R}} \right\rrbracket_1 q \bmod^{\llbracket\rrbracket} \mathtt{R}' \right) \right) \bmod^{\llbracket\rrbracket} \mathtt{R}'.$$

Once $\left\llbracket \frac{a\llbracket b\mathtt{R}/q\rrbracket_0}{\mathtt{R}} \right\rrbracket_1$ is computed, all the remaining multiplications are modular multiplications defined by the modular reduction $\bmod^{\llbracket\rrbracket} \mathtt{R}'$, and hence, we do not need the integer product $ab \in \mathbb{Z}$. In practice, we usually choose $\mathtt{R}' = \mathtt{R}$ for efficient computations. See Section 3.6 for the computation of $\left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket_0$.

Historical review. For unsigned arithmetic, [Bar86] proposed the case $b = 1$, and [Sho] proposed Barrett multiplication for b an integer. The signed version and its correspondence to Montgomery multiplication were discovered by [BHK$^+$21]. [BHK$^+$22, Section 2.4] improved the output range for $b \neq 1$ while increasing the precision of $\mathtt{R}$, and [HKS23] furthered the approximation nature of $\llbracket\rrbracket_1$ and improved the modular multiplications on microcontrollers. See [Dhe03] for a polynomial version.

3.5 Plantard Multiplication

[Pla21] proposed an unsigned modular multiplication essentially with precision $2\log_2 \mathtt{R}$. The signed versions were later proposed by [AMOT22, HZZ$^+$22]. Montgomery multiplication computes a representative of $ab \bmod q$ with absolute value bounded by $\frac{|a|\left|\bmod^{\llbracket\rrbracket_0} q\right| + \left|\bmod^{\llbracket\rrbracket_1} \mathtt{R}\right| q}{\mathtt{R}}$ when $|b| \leq \left|\bmod^{\llbracket\rrbracket_0} q\right|$. If we replace $\mathtt{R}$ with $\mathtt{R}^2$ and compute with

$$\frac{a\left(b\mathtt{R}^2 \bmod^{\llbracket\rrbracket_0} q\right) + \left(a\left(b\mathtt{R}^2 \bmod^{\llbracket\rrbracket_0} q\right)\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}^2\right) q}{\mathtt{R}^2},$$

we have the bound

$$\frac{|a|\left|\bmod^{\llbracket\rrbracket_0} q\right| + \left|\bmod^{\llbracket\rrbracket_1} \mathtt{R}^2\right| q}{\mathtt{R}^2}.$$

For signed arithmetic with $\left|\bmod^{\llbracket\rrbracket_1} \mathtt{R}^2\right| \leq \frac{\mathtt{R}^2}{2}$ and $\left|\bmod^{\llbracket\rrbracket_0} q\right| \leq \frac{q}{2}$, the bound is $\frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}^2}\right)$. In practice, we usually have $2 \nmid q$, $|a| \leq \mathtt{R}$, and $q < \mathtt{R}$, so the result is strictly smaller than $\frac{q}{2}$, implying

$$\frac{a\left(b\mathtt{R}^2 \bmod^{\llbracket\rrbracket_0} q\right) + \left(a\left(b\mathtt{R}^2 \bmod^{\llbracket\rrbracket_0} q\right)\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}^2\right) q}{\mathtt{R}} \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}.$$

We borrow the integer-approximation view from [HKS23] and proceed with [Pla21]'s innovation for implementing the above observation.

Definition 24 (Plantard reduction for integers, [Pla21, Hwa24a] and this thesis)**.** Let $\llbracket\rrbracket_1$, $\llbracket\rrbracket_2$, and $\llbracket\rrbracket_3$ be integer approximations, q and $\mathtt{R}$ be coprime integers greater than 1, $\tilde{\mathtt{R}}$ be a factor of $\mathtt{R}$, and $\mathtt{B}$ be an unspecified positive integer. If for all integers z with $|z| \leq \mathtt{B}$, we have

$$
\begin{aligned}
&\frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} \\
= &\left\llbracket \frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} - \frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \bmod^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3,
\end{aligned}
$$

then Plantard reduction computes a representative of $z\mathtt{R}^{-1}$ mod$q$ as

$$
\left\llbracket \frac{\left\llbracket \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) / \tilde{\mathtt{R}} \right\rrbracket_2 q}{\mathtt{R}/\tilde{\mathtt{R}}} \right\rrbracket_3 .
$$

Well-definedness of Definition 24.

$$
\begin{aligned}
&\left\llbracket \frac{\left\llbracket \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) / \tilde{\mathtt{R}} \right\rrbracket_2 q}{\mathtt{R}/\tilde{\mathtt{R}}} \right\rrbracket_3 \\
= &\left\llbracket \frac{\left(\left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) - \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \bmod^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right)\right) q \big/ \tilde{\mathtt{R}}}{\mathtt{R}/\tilde{\mathtt{R}}} \right\rrbracket_3 \\
= &\left\llbracket \frac{\left(\left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) - \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \bmod^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right)\right) q}{\mathtt{R}} \right\rrbracket_3 \\
= &\left\llbracket \frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} - \frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \bmod^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3 \\
= &\frac{z + \left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} .
\end{aligned}
$$

□

Definition 25 (Plantard reduction for positive integers, [Pla21, Hwa24a] and this thesis)**.** Let $\llbracket\rrbracket_1$, $\llbracket\rrbracket_2$, and $\llbracket\rrbracket_3$ be integer approximations, q and $\mathtt{R}$ be coprime integers greater than 1, $\tilde{\mathtt{R}}$ be a factor of $\mathtt{R}$, and $\mathtt{B}$ be an unspecified positive integer. If for all integers z with $0 \leq z \leq \mathtt{B}$, we have

$$
\begin{aligned}
& \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} \\
= \quad & \left\llbracket \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} + \frac{z - \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \ \mathrm{mod}^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3,
\end{aligned}
$$

then Plantard reduction computes a representative of $-z\mathtt{R}^{-1} \ \mathrm{mod} q$ as

$$
\left\llbracket \frac{\left\llbracket \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) / \tilde{\mathtt{R}} \right\rrbracket_2 q}{\mathtt{R} \big/ \tilde{\mathtt{R}}} \right\rrbracket_3 .
$$

Proof.

$$
\begin{aligned}
& \left\llbracket \frac{\left\llbracket \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) / \tilde{\mathtt{R}} \right\rrbracket_2 q}{\mathtt{R} \big/ \tilde{\mathtt{R}}} \right\rrbracket_3 \\
= \quad & \left\llbracket \frac{\left(\left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) - \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \ \mathrm{mod}^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right)\right) q \big/ \tilde{\mathtt{R}}}{\mathtt{R} \big/ \tilde{\mathtt{R}}} \right\rrbracket_3 \\
= \quad & \left\llbracket \frac{\left(\left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) - \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \ \mathrm{mod}^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right)\right) q}{\mathtt{R}} \right\rrbracket_3 \\
= \quad & \left\llbracket \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} - \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \ \mathrm{mod}^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3 \\
= \quad & \left\llbracket \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} + \frac{z - \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \ \mathrm{mod}^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3 \\
= \quad & \frac{-z + \left(zq^{-1} \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}}.
\end{aligned}
$$

□

Determining the quality of signed Plantard multiplication. We first analyze the quality of Plantard multiplication. As shown in the above proofs, the absolute values of the results are bounded by $\frac{\mathtt{B}+\left| \ \mathrm{mod}^{\llbracket\rrbracket_1} \mathtt{R} \right| q}{\mathtt{R}}$. In practice, $\llbracket\rrbracket_1$

is commonly chosen as $\lfloor\rceil$ so the result is upper-bounded by $\frac{q}{2} + \frac{\mathtt{B}}{\mathtt{R}}$. If $\frac{\mathtt{B}}{\mathtt{R}} < \frac{1}{2}$, then the result is an integer in $\left[-\frac{q}{2}, \frac{q}{2}\right)$. This is the case when z is a product of two arbitrary integers in $\left[-\frac{\sqrt{\mathtt{R}}}{2}, \frac{\sqrt{\mathtt{R}}}{2}\right]$. The same holds when one of the inputs is relaxed to an integer in $\left(-\sqrt{\mathtt{R}}, \sqrt{\mathtt{R}}\right)$.

Determining $\llbracket\rrbracket_2$ and $\llbracket\rrbracket_3$. Next, we analyze the sufficiency of the integer approximations $\llbracket\rrbracket_2$ and $\llbracket\rrbracket_3$. For simplicity, we analyze the case $\llbracket\rrbracket_3 = r \mapsto \lfloor r + \epsilon\rfloor$ for a positive real number $\epsilon \leq \frac{1}{2}$. For the correctness of Definition 24, $\epsilon > \frac{\mathtt{B}+\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|q}{\mathtt{R}}$ suffices and this covers the proposals by [AMOT22, HZZ$^+$22]. Compare this to [AMOT22, Theorem 3] and [HZZ$^+$22, Theorem 1]. If $\llbracket\rrbracket_2 = \lfloor\rceil$ and z is a product of two arbitrary integers in $\left[-\frac{\sqrt{\mathtt{R}}}{2}, \frac{\sqrt{\mathtt{R}}}{2}\right)$, we have $\epsilon > \frac{1}{4} + \frac{\tilde{\mathtt{R}}q}{2\mathtt{R}}$. As for the correctness of Definition 25, $\epsilon > \frac{\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|q}{\mathtt{R}}$ suffices and this covers the original proposal by [Pla21]. Compare this to [Pla21, Theorem 1].

On the necessary conditions for the sufficiency of our analyses. In previous two paragraphs, we analyze the sufficient conditions for the quality of signed Plantard multiplication and its correctness. We summarize some necessary conditions implied by the sufficient conditions. Assuming $\llbracket\rrbracket_3 = r \mapsto \lfloor r + \epsilon\rfloor$ for a positive real number $\epsilon \leq \frac{1}{2}$, we require $\frac{\mathtt{B}+\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|q}{\mathtt{R}} < \frac{1}{2}$ for the existence of ϵ in Definition 24. This implies $\mathtt{B} < \frac{\mathtt{R}}{2} - \left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|q < \frac{\mathtt{R}}{2}$ and $\frac{z+\left(z\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1}\mathtt{R}\right)q}{\mathtt{R}} \in \left[-\frac{q}{2}, \frac{q}{2}\right)$ when $\llbracket\rrbracket_1 = \lfloor\rceil$. Moving $\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|$ to the left-hand side, we require $\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right| < \frac{\mathtt{R}-2\mathtt{B}}{2q}$ and $q < \frac{\mathtt{R}-2\mathtt{B}}{2\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|}$. This covers the proposals by [AMOT22, HZZ$^+$22]. If z is a product of two arbitrary integers in $\left[-\frac{\sqrt{\mathtt{R}}}{2}, \frac{\sqrt{\mathtt{R}}}{2}\right)$, we must have $\frac{\mathtt{R}}{4} \leq \mathtt{B}$ and $\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right| < \frac{\mathtt{R}}{4q}$. If further $\llbracket\rrbracket_2 = \lfloor\rceil$, we require $\tilde{\mathtt{R}} < \frac{\mathtt{R}}{2q}$. As for Definition 25, we require $\frac{\left|\bmod^{\llbracket\rrbracket_2}\tilde{\mathtt{R}}\right|q}{\mathtt{R}} < \epsilon < \frac{\mathtt{R}-\mathtt{B}}{\mathtt{R}}$ and this covers the proposal by [Pla21].

When z is a product of two integers a and b with $|ab| < \frac{\mathtt{R}}{2}$, an obvious way to compute a representative of $ab \bmod q$ is to apply Plantard reduction to ab. Suppose b is known, we can further optimize the computation as follows.

Definition 26 (Plantard multiplication, [Pla21, AMOT22, HZZ$^+$22, Hwa24a] and this thesis)**.** Let $\llbracket\rrbracket_0$, $\llbracket\rrbracket_1$, $\llbracket\rrbracket_2$, and $\llbracket\rrbracket_3$ be integer approximations, q and $\mathtt{R}$ be coprime integers greater than 1, $\tilde{\mathtt{R}}$ be a factor of $\mathtt{R}$, $\mathtt{B}$ be an unspecified positive integer, and b be an integer with $|b'| < \mathtt{B}$ where $b' = b\mathtt{R} \bmod^{\llbracket\rrbracket_0} q$. If for all integers a with $|ab'| \leq \mathtt{B}$, we have

$$
\begin{aligned}
&\frac{ab' + \left(ab'\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} \\
= \quad &\left\llbracket \frac{ab' + \left(ab'\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) q}{\mathtt{R}} - \frac{ab' + \left(ab'\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R} \bmod^{\llbracket\rrbracket_2} \tilde{\mathtt{R}}\right) q}{\mathtt{R}} \right\rrbracket_3 ,
\end{aligned}
$$

then Plantard multiplication computes a representative of $ab \bmod q$ as

$$
\left\llbracket \frac{\left\llbracket \left(a\left(b'\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}\right) / \tilde{\mathtt{R}} \right\rrbracket_2 q}{\mathtt{R} / \tilde{\mathtt{R}}} \right\rrbracket_3
$$

with the precomputed value $b'\left(-q^{-1}\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}$.

Historical review. Fix $\tilde{\mathtt{R}} = \sqrt{\mathtt{R}}$. [Pla21] proposed the unsigned Plantard multiplication where $\llbracket\rrbracket_0 = \llbracket\rrbracket_1 = \llbracket\rrbracket_2 = \lfloor\rfloor$, and [AMOT22, HZZ$^+$22] proposed the signed versions where $\llbracket\rrbracket_0 = \llbracket\rrbracket_1 = \lfloor\rceil$. The primary difference of [AMOT22] and [HZZ$^+$22] is the implementation of $\frac{a\left(b'\left(-q^{-1}\right) \bmod^{\pm} \mathtt{R}\right) \bmod^{\pm} \mathtt{R}}{\sqrt{\mathtt{R}}}$: for an $a \in \left[-\frac{\sqrt{\mathtt{R}}}{2}, \frac{\sqrt{\mathtt{R}}}{2}\right)$, [AMOT22] computed the numerator with a multiplication of precision $\log_2 \mathtt{R} = \log_2 \mathtt{R} \times \log_2 \mathtt{R}$ whereas [HZZ$^+$22] computed the numerator with a multiplication of precision $\log_2 \mathtt{R} = \log_2 \sqrt{\mathtt{R}} \times \log_2 \mathtt{R}$.

3.6 Floor and Round of Fractions

This section goes through an application of modular multiplications to the floor and round of fractions. For an integer approximation $\llbracket\rrbracket$, an integer a, and a positive integer $q > 1$, we wish to compute $\left\llbracket \frac{a}{q} \right\rrbracket$. For the remainder of this section, we only discuss the floor function $\lfloor\rfloor$ and the round-half-up function $\lfloor\rceil$. Let $\llbracket\rrbracket_0$ and $\llbracket\rrbracket_1$ be either $\lfloor\rfloor$ or $\lfloor\rceil$. We have $a \bmod^{\llbracket\rrbracket_0} q = a - \left\llbracket \frac{a}{q} \right\rrbracket_0 q$ so $\left\llbracket \frac{a}{q} \right\rrbracket_0 = \frac{a - \left(a \bmod^{\llbracket\rrbracket_0} q\right)}{q}$. Suppose there is a positive integer $\mathtt{R}' > 1$ coprime to q and an integer approximation $\llbracket\rrbracket_1$ such that $\left\llbracket \frac{a}{q} \right\rrbracket_0 \in \operatorname{Img}\left(\bmod^{\llbracket\rrbracket_1} \mathtt{R}'\right)$ for all interested integers a, then we have

$$
\left\llbracket \frac{a}{q} \right\rrbracket_0 = q^{-1}\left(a - \left(a \bmod^{\llbracket\rrbracket_0} q\right)\right) \bmod^{\llbracket\rrbracket_1} \mathtt{R}'.
$$

Given the knowledge of $q^{-1} \operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'$, we can compute $\left[\!\!\left[\frac{a}{q}\right]\!\!\right]_0$ with modular reductions. Similarly, we also have

$$\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 = q^{-1}\left(ab - \left(ab \operatorname{mod}^{[\![\,]\!]_0} q\right)\right) \operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'$$

amounting to a modular multiplication $ab \operatorname{mod}^{[\![\,]\!]_0} q$ and a modular reduction $\operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'$ whenever $\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 \in \mathrm{Img}\left(\operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'\right)$. When q is an odd integer, we choose $\mathtt{R}'$ as a power of two for efficiency.

Theorem 2. Let $q, \mathtt{R}'$ be coprime integers greater than 1, $[\![\,]\!]_0, [\![\,]\!]_1$ be integer approximations with $[\![\,]\!]_0, [\![\,]\!]_1 = \lfloor\,\rfloor, \lfloor\,\rceil$. For integers a, b, we have

$$\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 \operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}' = q^{-1}\left(ab - ab \operatorname{mod}^{[\![\,]\!]_0} q\right) \operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'.$$

If $\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 \in \mathrm{Img}\left(\operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'\right)$, we have

$$\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 = q^{-1}\left(ab - ab \operatorname{mod}^{[\![\,]\!]_0} q\right) \operatorname{mod}^{[\![\,]\!]_1} \mathtt{R}'.$$

Selecting parameters for $[\![\,]\!]_0 = [\![\,]\!]_1 = \lfloor\,\rceil$ and odd $q > 0$. We go through the parameter selection when $[\![\,]\!]_0 = [\![\,]\!]_1 = \lfloor\,\rceil$ and $q > 0$ is an odd integer. We choose $\mathtt{R}'$ as the smallest power of two satisfying $|ab| < \frac{q(\mathtt{R}'-1)}{2}$. This ensures $\left\lfloor\frac{ab}{q}\right\rceil \in \mathrm{Img}\left(\operatorname{mod}^{\pm}\mathtt{R}\right)$. Since $\mathtt{R}$ is commonly a power of two and $\operatorname{mod}^{\pm}\mathtt{R}$ is very efficient in practice, it remains to implement $ab \operatorname{mod}^{\pm} q$ efficiently. One can choose any of Montgomery, Barrett, and Plantard multiplications with sufficient precision.

More generally, we have the following from modular multiplications.

Theorem 3. Let $a, b, \mathtt{R} > 1$ be integers and q be a positive odd integer. If $ab - \left\lfloor\frac{a\lfloor b\mathtt{R}/q\rfloor}{\mathtt{R}}\right\rceil q = ab \operatorname{mod}^{\pm} q$, we have $\left\lfloor\frac{ab}{q}\right\rceil = \left\lfloor\frac{a\lfloor b\mathtt{R}/q\rfloor}{\mathtt{R}}\right\rceil$.

Corollary 5. Let $a, b, \mathtt{R} > 1$ be integers and q be a positive odd integer. If $|a| < \frac{\mathtt{R}}{2\left|b\mathtt{R} \operatorname{mod}^{\pm} q\right|}$ then $\left\lfloor\frac{ab}{q}\right\rceil = \left\lfloor\frac{a\lfloor b\mathtt{R}/q\rfloor}{\mathtt{R}}\right\rceil$.

Chapter 4

Fast Homomorphisms

This chapter goes through several homomorphisms implementing polynomial multiplications.

4.1 Karatsuba and Toom–Cook

Let k be a positive integer, and $\mathcal{I} = \{0, \dots, 2k-2\}$ and $\{s_i\}_{i\in\mathcal{I}} \subset \mathbb{Q} \cup \{\infty\}$ be finite sets. Karatsuba [KO62] and Toom–Cook [Too63] compute the size-$(2k-1)$ product of two size-k polynomials with the maps $R[x]_{<k} \hookrightarrow R[x]/\left\langle \prod_{i\in\mathcal{I}}(x - s_i) \right\rangle \cong \prod_{i\in\mathcal{I}} R[x]/\langle x - s_i \rangle$. We call the composition of the maps **Toom-k** and $\{s_i\}_{i\in\mathcal{I}}$ the corresponding point set. For the case $k = 2$, we call the composition **Karatsuba**. [KO62] proposed the case $k = 2$ with the point set $\{0, 1, \infty\}$, [Too63] chose $k \geq 2$ and $\{s_i\}_{i\in\mathcal{I}} \subset \mathbb{Z}$, and [Win80] extended the choice of $\{s_i\}_{i\in\mathcal{I}}$ to $\mathbb{Q} \cup \{\infty\}$. Informally, evaluating x at ∞ refers to the extraction of the coefficients of the highest degree of polynomials and their product. For now, we illustrate the idea in the module view. Consider the following evaluation matrix implementing $R[x]_{<2} \to R[x]/\langle x \rangle \times R[x]/\langle x - 1 \rangle \times R[x]/\langle x - \infty \rangle$:

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

We define its "inverse" as the inverse of the map $R[x]/\langle x(x-1)(x-\infty)\rangle \to R[x]/\langle x\rangle \times R[x]/\langle x-1\rangle \times R[x]/\langle x-\infty\rangle$:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}^{-1} .$$

See Section 6.1 for a formal treatment of "evaluation at ∞." For a non-zero integer c, evaluating x at c^{-1} means mapping a polynomial $\boldsymbol{a}(x)$ to $c^{\deg(\boldsymbol{a})}\boldsymbol{a}(c^{-1})$ instead of $\boldsymbol{a}(c^{-1})$. Similarly, for integers a and $b \neq 0$, evaluating x at $\frac{a}{b}$ means mapping a polynomial $\boldsymbol{a}(x)$ to $b^{\deg(\boldsymbol{a})}\boldsymbol{a}\left(\frac{a}{b}\right)$ instead of $\boldsymbol{a}\left(\frac{a}{b}\right)$. See Section 5.1 for a more systematic treatment.

Large-dimensional transformations. Toom-k can be used for multiplying large-dimensional polynomials. For $n = kh$ a multiple of k, we have

$$R[x]_{<n} \cong \left(R[x] \Big/ \left\langle x^{\frac{n}{k}} - y \right\rangle [y] \right)_{<k} \hookrightarrow \prod_{i \in \mathcal{I}} \left(R[x] \Big/ \left\langle x^{\frac{n}{k}} - y \right\rangle \right) [y] \Big/ \langle y - s_i \rangle .$$

For polynomial multiplications in $R[x] \Big/ \left\langle x^{\frac{n}{k}} - y \right\rangle$, we compute the size-$\left(\frac{2n}{k} - 1\right)$ product and reduce modulo $x^{\frac{n}{k}} - y$. When n is a power of k, we can apply Toom-k recursively and the overall complexity of polynomial multiplication is $O\left(n^{\log_k(2k-1)}\right)$.

4.2 Discrete Fourier Transform

This section reviews principal nth root of unity and discrete Fourier transform.

4.2.1 Principal nth Root of Unity

For a ring R, a positive integer n, and an nth root of unity ω_n, we call ω_n a **principal nth root of unity** if

$$\Phi_n(\omega_n) = 0.$$

Below we give a necessary condition that will be used for defining discrete Fourier transform.

Theorem 4. For a ring R, an element $\xi \in R$, and a positive integer n, we have

$$\Phi_n(\xi) = 0 \longrightarrow \left(\forall j = 1, \ldots, n-1, \sum_{0 \le i < n} \xi^{ij} = 0 \right) .$$

Lemma 2. For a positive integer n and a factor $j \neq n$ of n, $\Phi_n(x)$ is a factor of $\sum_{0 \leq i < \frac{n}{j}} x^{ij}$.

Proof.

$$\sum_{0 \leq i < \frac{n}{j}} x^{ij} = \frac{x^n - 1}{x^j - 1} = \frac{\prod_{d|n} \Phi_d(x)}{\prod_{d|j} \Phi_d(x)} = \Phi_n(x) \cdot \prod_{d|n, d \nmid j, d<n} \Phi_d(x).$$

Therefore, $\Phi_n(x)$ is a factor of $\sum_{0 \leq i < \frac{n}{j}} x^{ij}$.

□

Lemma 3. For a ring R, an element $\xi \in R$, a positive integer n, and a factor $j \neq n$ of n, $\Phi_n(\xi)$ is a factor of $\sum_{0 \leq i < \frac{n}{j}} \xi^{ij}$.

Proof. The proof immediately follows from applying the map $x \mapsto \zeta : R[x] \to R$ to Lemma 2. □

Proof of Theorem 4. For any $j = 1, \ldots, n-1$, we define $l = \gcd(j, n)$ and find

$$\sum_{0 \leq i < n} \xi^{ij} = l \sum_{0 \leq i < \frac{n}{l}} \xi^{ij} = l \sum_{0 \leq i < \frac{n}{l}} \xi^{il} = 0$$

according to Lemma 3.

□

Theorem 5. For a non-commutative ring R, Theorem 4 holds when ξ belongs to the center of R where the center is the subset consisting of elements commuting to all elements in R.

4.2.2 Discrete Fourier Transform

Discrete Fourier transform (DFT) is a special case of the Chinese remainder theorem (CRT) for polynomial rings. Let R be a ring, n be a positive integer, and $\omega_n \in R$ be a principal nth root of unity. The size-n DFT refers to the following algebra isomorphism:

$$\mathcal{F}_{\omega_n} : \begin{cases} \dfrac{R[x]}{\langle x^n - 1 \rangle} & \to & \prod_{0 \leq i < n} \dfrac{R[x]}{\langle x - \omega_n^i \rangle} \\ \boldsymbol{a}(x) & \mapsto & \left(\boldsymbol{a}\left(\omega_n^i\right)\right)_{0 \leq i < n} \end{cases}$$

with the inverse

$$\mathcal{F}_{\omega_n}^{-1} : \begin{cases} \prod_{0 \le i < n} \frac{R[x]}{\langle x - \omega_n^i \rangle} & \rightarrow & \frac{R[x]}{\langle x^n - 1 \rangle} \\ (\hat{a}_i)_{0 \le i < n} & \mapsto & \sum_{0 \le i < n} \boldsymbol{r}_i \hat{a}_i \end{cases}$$

where $\boldsymbol{r}_i := \frac{1}{n} \sum_{0 \le j < n} \omega_n^{-ij} x^j$. The correctness follows from the definition of a principal nth root of unity. One also finds $\mathcal{F}_{\omega_n}^{-1} = \mathcal{F}_{\omega_n^{-1}}$ as module isomorphisms.

Complexity. Obviously, a straightforward computation amounts to $n^2 - 2n + 1$ multiplications and $n^2 - n$ additions/subtractions: we need $n - 1$ additions/subtractions for $\boldsymbol{a}\left(\omega_n^0 = 1\right) = \sum_{0 \le j < n} 1$, and $n^2 - 2n + 1$ multiplications and $n^2 - 2n + 1$ additions/subtractions for $\left(\boldsymbol{a}\left(\omega_n^i\right)\right)_{1 \le i < n}$. One can replace $n - 1$ multiplications and $n - 1$ additions/subtractions with a single addition by exploiting $\sum_{0 \le i < n} \left(\boldsymbol{a}\left(\omega_n^i\right) - a_0\right) = 0$ [AHY22, Section 3.1.2]: we pick an i from $\{1, \ldots, n - 1\}$, and compute

$$\forall j \in \{0, \ldots, n - 1\} - \{i\}, \boldsymbol{a}\left(\omega_n^j\right) - a_0 = \sum_{1 \le k < n} a_k \omega_n^{jk}$$

followed by

$$\boldsymbol{a}\left(\omega_n^i\right) = a_0 + \left(\boldsymbol{a}\left(\omega_n^i\right) - a_0\right) = a_0 - \sum_{0 \le j < n, j \ne i} \left(\boldsymbol{a}\left(\omega_n^j\right) - a_0\right)$$

and

$$\forall j \in \{0, \ldots, n - 1\} - \{i\}, \boldsymbol{a}\left(\omega_n^j\right) = a_0 + \left(\boldsymbol{a}\left(\omega_n^j\right) - a_0\right).$$

If $n = 3$, we only need 1 multiplication for $\omega_3 (a_1 - a_2)$ and 7 additions/subtractions by replacing ω_3^2 with $-1 - \omega_3$ [Has22]. For the rest of the thesis, we adopt [AHY22]'s approach with $n^2 - 3n + 2$ multiplications and $n^2 - 2n + 2$ additions/subtractions for arbitrary size-n DFT.

Corollary 6. For a positive integer n, 2 is a principal nth root of unity defining a size-n cyclic DFT over $\mathbb{Z}_{\Phi_n(2)}$. This is the well-known Fermat number transform [SS71, AB74].

For an invertible element $\zeta \in R$, discrete weighted transform (DWT) generalizes DFT into an isomorphism $R[x]/\langle x^n - \zeta^n \rangle \cong \prod_{0 \le i < n} R[x]/\langle x - \zeta \omega_n^i \rangle$ with $\boldsymbol{r}_i := \frac{1}{n} \sum_{0 \le j < n} \zeta^{-j} \omega_n^{-ij} x^j$ in the inversion map [CF94]. We denote the isomorphisms as $\mathcal{F}_{\omega_n, \zeta}$ and $\mathcal{F}_{\omega_n, \zeta}^{-1}$ and call them cyclic when $\zeta^n = 1$ and negacyclic when $\zeta^n = -1$. Obviously, $\mathcal{F}_{\omega_n,\zeta\neq 1}$ requires $n-1$ additional multiplications compared to $\mathcal{F}_{\omega_n}$.

There are three conditions for defining an invertible DWT for $R[x]/\langle x^n - \zeta^n \rangle$:

- The positive integer n must be invertible in R. Notice that positive integers are encoded as repeat additions of the identity of R, and negative integers are encoded as repeat additions of the additive inverse of the identity of R.
- The element ζ must be invertible in R.
- There must exist a principal nth root of unity ω.

Historical review of the conditions. For defining a size-n DFT, [Pol71] showed that n must be a factor of $q-1$ if $R = \mathbb{F}_q$ and $p-1$ if $R = \mathbb{Z}_{p^k}$ for a prime p. The latter says that for $R = \mathbb{Z}_m$ with prime factorization $m = \prod_i p_i^{d_i}$, n must divide $\gcd(p_i - 1)$ [Pol71, AB74]. [DV78b, Theorem 4] gave the condition when R is a product of local rings[1], and [Für09, Section 2] showed that a principal nth root of unity suffices. The cyclotomic condition was used in [SS71] and stated in [Für09] for a power-of-two n. The proof in [Für09] naturally generalizes to a prime-power n. The cyclotomic condition, although obvious, does not appear in the literature at the best of author's knowledge.

Table 4.1 summarizes the number of arithmetic and conditions for $\mathcal{F}_{\omega_n}$ and $\mathcal{F}_{\omega_n,\zeta\neq 1}$.

Table 4.1: Summary of the number of arithmetic and conditions of $\mathcal{F}_{\omega_n}$ and $\mathcal{F}_{\omega_n,\zeta\neq 1}$.

	Arithmetic		Condition		
	# mul.	# add./sub.	$\exists\omega_n$	$\exists n^{-1}$	$\exists\zeta^{-1}$
$\mathcal{F}_{\omega_n}$	$n^2 - 3n + 2$	$n^2 - 2n + 2$	✓	✓	-
$\mathcal{F}_{\omega_n,\zeta\neq 1}$	$n^2 - 2n + 1$	$n^2 - 2n + 2$	✓	✓	✓

4.3 Cooley–Tukey Fast Fourier Transform

For the DFT implementing $R[x]/\langle x^n - \zeta^n \rangle \cong \prod_{0\leq i<n} R[x]/\langle x - \zeta\omega_n^i \rangle$, **Cooley–Tukey FFT** [CT65, CF94] improves the computation when n factors. Suppose

---

[1] A local ring is a ring with a unique maximal left/right-ideal.

$n = \prod_{0 \le j < h} n_j$. We define

$$\boldsymbol{g}_{i_0,\ldots,i_{h-1}} := x - \zeta \omega_n^{\sum_j i_j \prod_{l<j} n_l}$$

for all $0 \le i_j < n_j$ and find $x^n - \zeta^n = \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}}$. Since all the $\boldsymbol{g}_{i_0,\ldots,i_{h-1}}$'s are coprime, we have the following series of isomorphisms:

$$\frac{R[x]}{\left\langle \prod_{i_0,\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \prod_{i_0} \frac{R[x]}{\left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \cdots \cong \prod_{i_0,\ldots,i_{h-1}} \frac{R[x]}{\left\langle \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle}.$$

Complexity. For each $j = 0, \ldots, h-1$,

$$\prod_{i_0,\ldots,i_j} \frac{R[x]}{\left\langle \prod_{i_{j+1},\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle} \cong \prod_{i_0,\ldots,i_{j+1}} \frac{R[x]}{\left\langle \prod_{i_{j+2},\ldots,i_{h-1}} \boldsymbol{g}_{i_0,\ldots,i_{h-1}} \right\rangle}$$

amounts to $\frac{n}{n_j}$ size-n_j DFTs. We analyze the number of multiplications. If $\zeta \ne 1$, the total number of multiplications is

$$\sum_{0 \le i < h} \frac{n}{n_i} \left(n_i^2 - 2n_i + 1\right) = n \sum_{0 \le i < h} \left(n_i - 2 + \frac{1}{n_i}\right).$$

If $\zeta = 1$ and we apply the cyclic DFTs whenever possible, the total number of multiplications is

$$\begin{aligned} & \sum_{0 \le i < h} \left(\frac{n \Big/ \prod_{j<i} n_j}{n_i} \left(n_i^2 - 3n_i + 2\right) + \frac{n - n \Big/ \prod_{j<i} n_j}{n_i} \left(n_i^2 - 2n_i + 1\right) \right) \\ = \quad & n \sum_{0 \le i < h} \left(n_i - 2 + \frac{1}{n_i}\right) - \sum_{0 \le i < h} \frac{n\,(n_i - 1)}{\prod_{j \le i} n_j} \\ = \quad & n \sum_{0 \le i < h} \left(n_i - 2 + \frac{1}{n_i}\right) - (n-1). \end{aligned}$$

Furthermore, we need $n \sum_{0 \le i < h} \left(n_i - 2 + \frac{2}{n_i}\right)$ additions/subtractions for both $\mathcal{F}_{\omega_n}$ and $\mathcal{F}_{\omega_n, \zeta \ne 1}$.

A small example. We give a small example for $R[x]/\left\langle x^4 - 1\right\rangle$. Suppose 4 is invertible in R and there is a principal 4th root of unity $\omega_4 \in R$. We write the map $\boldsymbol{a}(x) \mapsto \left(\boldsymbol{a}(1), \boldsymbol{a}\left(\omega_4\right), \boldsymbol{a}\left(\omega_4^2\right), \boldsymbol{a}\left(\omega_4^3\right)\right)$ as

$$\mathcal{F}_{\omega_4} = P_{4:(12)} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & \omega_4 & -1 & -\omega_4 \\ 1 & -\omega_4 & -1 & \omega_4 \end{pmatrix} = P_{4:(12)} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & \omega_4 \\ 0 & 0 & 1 & -\omega_4 \end{pmatrix} \left(\mathcal{F}_{-1} \otimes I_2\right)$$

where $P_{4:(12)}$ is the permutation matrix swapping the 1st and the 2nd elements drawn from a set of 4 elements. Obviously, each of $\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & \omega_4 \\ 0 & 0 & 1 & -\omega_4 \end{pmatrix}$ and $(\mathcal{F}_{-1} \otimes I_2)$ can be implemented with linearly number of arithmetic in R.

4.4 Bruun's Fast Fourier Transform

After the introduction of Cooley–Tukey FFT for the complex case $R = \mathbb{C}$, many works proposed several optimizations for the real inputs. [Bru78] proposed **Bruun's FFT** for the power-of-two case, [DH84] proposed split-radix FFT, [Bra84] proposed fast Hartley transform for the discrete Hartley transform (DHT) [Har42][2], [Mur96] generalized Bruun's FFT to arbitrary even sizes, and [JF07, Ber07, LVB07] improved the split-radix FFT.

This section reviews the works [Bru78, Mur96] over complex numbers for historical reasons. However, the actual use case relevant to us are the factorization of cyclotomic polynomials over finite fields [BC87, BGM93, Mey96]. See [TW13, BMGVdO15, WYF18, WY21] for recent progresses on this topic.

The complex case. Let $n = \prod_j n_j$ be a positive integer, $\xi, \zeta \in \mathbb{C}$ be invertible elements, and $\omega_n \in \mathbb{C}$ be a principal nth root of unity. Bruun's FFT chooses $\boldsymbol{g}_{i_0,\dots,i_{h-1}}$ as follows:

$$\boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^2 - \left(\xi \omega_n^{\sum_j i_j \prod_{l<j} n_l} + \xi^{-1} \omega_n^{-\sum_j i_j \prod_{l<j} n_l} \right) \zeta x + \zeta^2.$$

If $\boldsymbol{g}_{i_0,\dots,i_{h-1}}$'s are coprime ($\xi \neq \xi^{-1}$ in the complex case), we have a fast transformation $R[x]/\langle x^{2n} - \left(\xi^n + \xi^{-n}\right) \zeta^n x^n + \zeta^{2n} \rangle$ since $\prod_{i_0,\dots,i_{h-1}} \boldsymbol{g}_{i_0,\dots,i_{h-1}} = x^{2n} - \left(\xi^n + \xi^{-n}\right) \zeta^n x^n + \zeta^{2n}$. For $\zeta = 1, \xi = \omega_{4n} \in \mathbb{C}$, this implements the isomorphism $\mathbb{C}[x]/\langle x^{2n} + 1 \rangle \cong \prod_i \mathbb{C}[x]/\langle x - \omega_{4n}^{1+2i} \rangle$ if we further split into linear factors.

The finite field cases. In this paper, we are interested in the case $R = \mathbb{F}_q$ with $q \equiv 3 \pmod 4$ which relies on the following theorem from [BGM93]:

Theorem 6 ([BGM93]). Let $q \equiv 3 \pmod 4$ be a prime and 2^w be the largest power-of-two factor of $q + 1$. For $k < w$, $x^{2^k} + 1$ factors into irreducible trinomials

---

[2] One can derive DFT and DHT from each other with linearly number of arithmetic during post-processing. Therefore, improvement for one of them transfers to the other one.

$x^2 + \gamma x + 1 \in \mathbb{F}_q[x]$. For $k \geq w$, $x^{2^k} + 1$ factors into irreducible trinomials $x^{2^{k-w+1}} + \gamma x^{2^{k-w}} - 1 \in \mathbb{F}_q[x]$.

4.5 Good–Thomas Fast Fourier Transform

Good–Thomas FFT exploits the factorization of $n = \prod_{0 \leq j < d} n_j$ when n_j's are coprime to each other [Goo58]. Let ω_n be a principal nth root of unity and $(e_j)_{0 \leq j < d}$ be the unique tuple of positive integers satisfying $1 \equiv \sum_{0 \leq j < n} e_j \pmod{n}$. Consider the set of principal roots of unity $\left\{\omega_{n_j} \coloneqq \omega_n^{e_j}\right\}_{0 \leq j < d}$, we have the identity $\omega_n = \prod_{0 \leq j < d} \omega_{n_j}$. We explain Good–Thomas FFT below with different levels of abstractions.

Coefficient view. For a polynomial $\sum_{0 \leq i < n} a_i x^i$, we rewrite the kth component of its image under $\mathcal{F}_n$ as

$$\sum_{0 \leq i < n} a_i \omega_n^{ik} = \sum_{i_0} \cdots \sum_{i_{d-1}} a_{\sum_{0 \leq j < d} e_j i_j} \prod_{0 \leq j < d} \omega_{n_j}^{i_j k_j}$$

where $i_j = i \bmod n_j$ and $k_j = k \bmod n$, and find the right-hand side a multi-dimensional cyclic DFT.

Homomorphism view. In terms of homomorphisms, we have

$$\mathcal{F}_{\omega_n} = \pi^{-1} \circ \left(\bigotimes_{0 \leq j < d} \mathcal{F}_{\omega_{n_j}} \right) \circ \pi$$

where π is the permutation induced by the additive group isomorphism $1 \mapsto \left(\underbrace{1, \ldots, 1}_{d} \right)$ from $\mathbb{Z}_n$ to $\prod_{0 \leq j < d} \mathbb{Z}_{n_j}$.

Algebra view. Consider the group algebra $R[G]$ with a group isomorphism $G \cong \prod_{0 \leq j < d} G_j$. We have $R[G] \cong \bigotimes_{0 \leq j < d} R[G_j]$. For $G = \mathbb{Z}_n$, we have the following commutative diagram.

Figure 4.1: Commutative diagram of Good–Thomas FFT in the group algebra view.

$$\begin{tikzcd}
R\left[\mathbb{Z}_n\right] \arrow[r, "\pi"] \arrow[d, "\mathcal{F}_{\omega_n}"'] & \bigotimes_{0\le j<d} R\left[\mathbb{Z}_{n_j}\right] \arrow[d, "\bigotimes_{0\le j<d} \mathcal{F}_{\omega_{n_j}}"] \\
\mathrm{Img}\left(\mathcal{F}_{\omega_n}\right) & \mathrm{Img}\left(\bigotimes_{0\le j<d} \mathcal{F}_{\omega_{n_j}}\right) \arrow[l, "\pi^{-1}"]
\end{tikzcd}$$

Polynomial ring view. In the language of polynomial rings, the cyclic size-n DFT is implemented as the following multi-dimensional cyclic DFT:

$$\begin{aligned}
\frac{R[x]}{\langle x^n-1\rangle} &\cong \frac{R[x_0,\dots,x_{d-1}]}{\left\langle x-\prod_j x_j, x_0^{n_0}-1,\dots,x_{d-1}^{n_{d-1}}-1\right\rangle} \\
&\cong \prod_{i_0,\dots,i_{d-1}} \frac{R[x_0,\dots,x_{d-1}]}{\left\langle x-\prod_j x_j, x_0-\omega_{n_0}^{i_0},\dots,x_{d-1}-\omega_{n_{d-1}}^{i_{d-1}}\right\rangle} \\
&\cong \prod_i \frac{R[x]}{\langle x-\omega_n^i\rangle}.
\end{aligned}$$

Complexity. Suppose each of $\mathcal{F}_{\omega_{n_j}}$ requires $n_j^2-3n_j+2$ multiplications. We rewrite $\bigotimes_{0\le j<d}\mathcal{F}_{\omega_{n_j}}$ as a composition of

$$\mathrm{id}_{\prod_{k<j} n_k} \otimes \mathcal{F}_{\omega_{n_j}} \otimes \mathrm{id}_{\prod_{k>j} n_k}$$

for all $j=0,\dots,d-1$, and implement each with $n\left(n_j-3+\frac{2}{n_j}\right)$ multiplications. This requires $n\sum_{0\le j<d}\left(n_j-3+\frac{2}{n_j}\right)$ multiplications in total. As for the number of additions/subtractions, it is the same as the Cooley–Tukey FFT.

A small example. Consider the isomorphism $\mathcal{F}_{\omega_6}$. We define $P_{6:(14)}$ as the permutation matrix swapping the 1st and the 4th element drawn from a set of 6 elements. $P_{6:(14)}$ implements the permutation mapping the one-dimensional indices $\{0,\dots,5\}$ to the row-major representation of the two-dimensional indices

$$\left\{\begin{matrix}(0 \bmod 3, 0 \bmod 2), & (4 \bmod 3, 4 \bmod 2), & (2 \bmod 3, 2 \bmod 2), \\ (3 \bmod 3, 3 \bmod 2), & (1 \bmod 3, 1 \bmod 2), & (5 \bmod 3, 5 \bmod 2)\end{matrix}\right\}.$$

We now rewrite $\mathcal{F}_{\omega_6}$ as

$$\mathcal{F}_{\omega_6} = P_{6:(14)}^{-1} \left(\mathcal{F}_{-1} \otimes \mathcal{F}_{\omega_6^4} \right) P_{6:(14)}$$

and implement $\mathcal{F}_{-1} \otimes \mathcal{F}_{\omega_6^4}$ as $(\mathcal{F}_{-1} \otimes I_3) \left(I_2 \otimes \mathcal{F}_{\omega_6^4} \right)$.

4.6 Vector–Radix Fast Fourier Transform

We know that one-dimensional size-n cyclic convolution can be turned into a multi-dimensional cyclic convolution based on a coprime factorization of n. If we apply DFTs to each dimensions and cache the results, then we save the cost of transformation significantly. This section explains how to save more multiplications by directly optimizing a multi-dimensional transform with **vector-radix FFT** [HMCS77].

Suppose we have a tensor product of homomorphisms $\otimes_j f_j$. A crucial property while tensoring two compositions $f_{0,0} \circ f_{0,1}$ and $f_{1,0} \circ f_{1,1}$ is that $(f_{0,0} \circ f_{0,1}) \otimes (f_{1,0} \circ f_{1,1}) = (f_{0,0} \otimes f_{1,0}) \circ (f_{0,1} \otimes f_{1,1})$. Usually, f_j can be characterized as a composition of multiplicative steps and additive steps. During the multiplicative steps, we only multiply coefficients by some constants. For the additive steps, we perform additions and subtractions. The key is that multiplicative steps are faster if we apply their composition directly.

A small example. Suppose we have two multiplicative steps with the matrix representations $M_0 = \begin{pmatrix} 1 & 0 \\ 0 & \zeta_0 \end{pmatrix} \otimes I_2$ and $M_1 = I_2 \otimes \begin{pmatrix} 1 & 0 \\ 0 & \zeta_1 \end{pmatrix}$. If we implement M_0 and M_1 separately, then we need four multiplications. Since $M_0 M_1$ is a diagonal matrix, we only need three multiplications for $M_0 M_1$.

4.7 Rader's Fast Fourier Transform

Let n be a positive integer, $\mathcal{I} = \{0, \dots, n-1\}$, and $\omega_n \in R$ be a principal nth root of unity. If n is an odd prime, **Rader's FFT** computes the map $\boldsymbol{a} \mapsto \left(\boldsymbol{a}(\omega_n^i) \right)_{i \in \mathcal{I}}$ with a size-$(n-1)$ cyclic convolution [Rad68].

Write $(a_j)_{j \in \mathcal{I}} \coloneqq \boldsymbol{a}$ and $(\hat{a}_i)_{i \in \mathcal{I}} \coloneqq \left(\boldsymbol{a}(\omega_n^i) \right)_{i \in \mathcal{I}}$, and let $\mathcal{I}^* \coloneqq \{1, \dots, n-1\}$ be an index set. Since n is prime, there is a $g \in \mathcal{I}$ with $\left\{ g^k \bmod n \in \mathcal{I} | k \in \mathbb{Z}_{n-1} \right\} = \mathcal{I}^*$. Consider the reindexing maps $j \in \mathcal{I}^* \mapsto -\log_g j \in \mathbb{Z}_{n-1}$ and $i \in \mathcal{I}^* \mapsto \log_g i \in \mathbb{Z}_{n-1}$ where $\log_g$ is the discrete logarithm, Rader's FFT splits the computation

$(a_j)_{j\in\mathcal{I}} \mapsto (\hat{a}_i)_{i\in\mathcal{I}}$ into $\hat{a}_0 = \sum_{j\in\mathcal{I}} a_j$ and $\hat{a}_i = a_0 + \sum_{j\in\mathcal{I}^*} a_j\omega_n^{ij}$ for $i \in \mathcal{I}^*$. For the cases $i \in \mathcal{I}^*$, we move a_0 to the left-hand side, and rewrite it as

$$\hat{a}_{g^{\log_g i}} - a_0 = \sum_{j\in\mathcal{I}^*} a_j\omega_n^{ij} = \sum_{-\log_g j\in\mathbb{Z}_{n-1}} a_{g^{\log_g j}}\omega_n^{g^{\log_g i+\log_g j}}.$$

We can now compute $\left(\hat{a}_{g^k} - a_0\right)_{k\in\mathbb{Z}_{n-1}}$ as the size-$(n-1)$ cyclic convolution of $\left(a_{g^{-k}}\right)_{k\in\mathbb{Z}_{n-1}}$ and $\left(\omega_n^{g^k}\right)_{k\in\mathbb{Z}_{n-1}}$.

A small example. We give an example for $n = 5$ and $g = 2$:

$$\begin{pmatrix} \hat{a}_{2^1} - a_0 \\ \hat{a}_{2^2} - a_0 \\ \hat{a}_{2^3} - a_0 \\ \hat{a}_{2^4} - a_0 \end{pmatrix} = \begin{pmatrix} a_{2^4}\omega_5^{2^1} + a_{2^3}\omega_5^{2^4} + a_{2^2}\omega_5^{2^3} + a_{2^1}\omega_5^{2^2} \\ a_{2^4}\omega_5^{2^2} + a_{2^3}\omega_5^{2^1} + a_{2^2}\omega_5^{2^4} + a_{2^1}\omega_5^{2^3} \\ a_{2^4}\omega_5^{2^3} + a_{2^3}\omega_5^{2^2} + a_{2^2}\omega_5^{2^1} + a_{2^1}\omega_5^{2^4} \\ a_{2^4}\omega_5^{2^4} + a_{2^3}\omega_5^{2^3} + a_{2^2}\omega_5^{2^2} + a_{2^1}\omega_5^{2^1} \end{pmatrix}.$$

4.8 Comparisons

We briefly compare Cooley–Tukey, Bruun, Good–Thomas, vector-radix, Rader, and Toom–Cook. Table 4.2 summarizes the domains and images, and Table 4.3 summarizes the defining conditions.

Cooley–Tukey, Good–Thomas, and vector-radix. Both Cooley–Tukey and Good–Thomas relies on a factorization of the dimension n, the existence of a principal nth root of unity, and the existence of n^{-1} in the coefficient ring. While Cooley–Tukey works for arbitrary factorization of n, Good–Thomas relies on a coprime factorization and can be combined with vector-radix FFT. As for the shape of polynomial modulus, Cooley–Tukey is definable on $R[x]/\langle x^n - \zeta^n\rangle$, and Good–Thomas is definable only on $R[x]/\langle x^n - 1\rangle$. Generally speaking, if the order of ζ is coprime to n, we can also define Good–Thomas on $R[x]/\langle x^n - \zeta^n\rangle$ via truncation as illustrated in [HVDH22, Sections 3.5 and 3.6]. If both approaches are definable, Good–Thomas saves linearly number of multiplications and combining with vector-radix FFT saves even more.

Bruun vs others. While Cooley–Tukey factors into polynomial rings with binomial moduli, Bruun factors into polynomial rings with trinomial moduli. If the coefficient ring is a finite field or finite ring, Bruun works in some cases where Cooley–Tukey does not since factoring into binomials implies factoring into trinomials but the converse does not always hold. The downside of Bruun is the increased number of arithmetic during the transformation.

Rader vs others. Rader converts size-n cyclic DFT into a size-$(n-1)$ cyclic convolution with linear pre- and post-processing when n is an odd prime. Other approaches rely on a factorization of n, implying that n must be composite.

Toom–Cook vs others. Cooley–Tukey, Bruun, Good–Thomas, vector-radix, and Rader are isomorphisms where the dimensions remain the same after the transformation. On the hand, Toom–Cook is a monomorphism where the dimension becomes larger after the transformation. For the definability, Toom–Cook requires the existences of the inverses of some integers. This is generally more favorable than the FFTs since one can always go for localization for constructing the inverses of integers, which, in practice, amounts to replacing the coefficient ring with a slightly larger one. A size-n FFT requires the existence of a principal nth root of unity and the inverse n^{-1} where the former only exists in certain coefficient ring.

Table 4.2: Overview of the domains and images of Cooley–Tukey, Bruun, Good–Thomas, vector-radix, Rader, and Toom–Cook.

Approach	Domain	Image
Cooley–Tukey	$\frac{R[x]}{\langle x^n-\zeta^n\rangle}$	$\prod_i \frac{R[x]}{\langle x-\zeta\omega_n^i\rangle}$
Good–Thomas	$\frac{R[x]}{\langle x^n-1\rangle}$	$\prod_i \frac{R[x]}{\langle x-\omega_n^i\rangle}$
Bruun	$\frac{R[x]}{\langle x^{2n}-(\xi^n+\xi^{-n})\zeta^n x^n+\zeta^{2n}\rangle}$	$\prod_i \frac{R[x]}{\left\langle x^2-\left(\xi\omega_n^i+(\xi\omega_n^i)^{-1}\right)\zeta x+\zeta^2\right\rangle}$
Rader	$\frac{R[x]}{\langle x^n-1\rangle}$	$\prod_i \frac{R[x]}{\langle x-\omega_n^i\rangle}$
Vector-radix	$\bigotimes_d \frac{R[x_d]}{\left\langle x_d^{n_d}-1\right\rangle}$	$\bigotimes_d \prod_{i_d} \frac{R[x_d]}{\left\langle x_d-\omega_{n_d}^{i_d}\right\rangle}$
Toom–Cook	$R[x]_{<n}$	$\prod_{i=0,\dots,2n-2} \frac{R[x]}{\langle x-s_i\rangle}$

Table 4.3: Overview of the defining conditions of Cooley–Tukey, Bruun, Good–Thomas, vector-radix, Rader, and Toom–Cook.

Approach	Condition
Cooley–Tukey	1. $\exists \omega_n, \zeta, \zeta^{-1}, n^{-1} \in R$. 2. $\exists$ a factorization of n.
Good–Thomas	1. $\exists \omega_n, n^{-1} \in R$. 2. $\exists$ a coprime factorization of n.
Bruun	$\exists \xi\omega_n^i + \left(\xi\omega_n^i\right)^{-1}, \zeta, \zeta^{-1}, n^{-1} \in R$.
Rader	1. $\exists \omega_n, n^{-1} \in R$. 2. Odd prime n.
Vector-radix	$\forall d, \exists \omega_{n_d}, n_d^{-1} \in R$.
Toom–Cook	Inverses of integers.

Chapter 5

Coefficient Ring Embedding

While choosing an isomorphism, the defining conditions might not hold. This chapter goes through several coefficient ring embedding techniques adjoining the defining conditions.

5.1 Localization

For a ring R and a multiplicative set $S \subset R$, localization is a standard technique introducing inverses of S. In this paper, we are interested in the case when R is a polynomial ring over $\mathbb{Z}_{2^k}$ and S is the set $\left\{2^k | k \in \mathbb{Z}_{\geq 0}\right\}$.

Multiplicative set. For a subset S of a ring, we call it a **multiplicative set** if it is closed under the ring multiplication. For example, $\left\{z^k | k \in \mathbb{Z}_{\geq 0}\right\} \subset \mathbb{Z}$ is a multiplicative set for any $z \in \mathbb{Z}$. We denote $\left\{z^k | k \in \mathbb{Z}_{\geq 0}\right\}$ as $z^{\mathbb{Z}_{\geq 0}}$.

Localization of a ring. For a ring R and a multiplicative set $S \subset R$, localization formally introduces divisions by elements in S. Consider the set $R \times S$ and the following equivalence relation

$$\forall (r_1, s_1), (r_2, s_2) \in R \times S, (r_1, s_1) \sim (r_2, s_2) \longleftrightarrow \exists s \in S, s s_2 r_1 = s s_1 r_2.$$

The **localization of R at S** is defined as the quotient ring $S^{-1}R \coloneqq R \times S/\sim$. The most common example is the set of rational numbers $\mathbb{Q}$ – we define $\mathbb{Q}$ as the localization of $\mathbb{Z}$ at $\mathbb{Z}_{>0}$. Another example is the set of dyadic rational numbers $2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}$:

$$2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z} \coloneqq \left(2^{\mathbb{Z}_{\geq 0}}\right)^{-1} \mathbb{Z} = \left\{\frac{a}{2^k} | a \in \mathbb{Z}, k \in \mathbb{Z}_{\geq 0}\right\}.$$

Inverting a monomorphism. Let $\mathcal{A}$ and $\mathcal{B}$ be rings and $\eta : \mathcal{A} \to \mathcal{B}$ be a ring monomorphism. For an integer $z \in \mathbb{Z} - \{0\}$, suppose we find a map $\psi_z : \eta(\mathcal{A}) \to \mathcal{A}$ such that

$$\forall \boldsymbol{a} \in \mathcal{A}, (\psi_z \circ \eta)(\boldsymbol{a}) = z\boldsymbol{a}.$$

We define a homomorphism $\xi : \mathcal{Z}^{-1}\eta(\mathcal{A}) \to \mathcal{Z}^{-1}\mathcal{A}$ as

$$\forall z^{-k}\eta(\boldsymbol{a}) \in \mathcal{Z}^{-1}\eta(\mathcal{A}), \xi\left(z^{-k}\eta(\boldsymbol{a})\right) := z^{-1-k}\psi_z(\eta(\boldsymbol{a})).$$

If we restrict the image of ξ to $\eta(\mathcal{A})$, we find $\xi|_{\eta(\mathcal{A})} := \left(\eta(\boldsymbol{a}) \mapsto z^{-1}\psi_z(\eta(\boldsymbol{a}))\right) = \eta^{-1}$. In summary, to invert η while given ψ_z with $z \in \mathbb{Z} - \{0\}$ non-invertible in $\mathcal{A}$, it suffices to define $\xi : \mathcal{Z}^{-1}\eta(\mathcal{A}) \to \mathcal{Z}^{-1}\mathcal{A}$ and apply $\xi|_{\eta(\mathcal{A})}$.

A practically important example. We illustrate localization with the example $\mathbb{Z}_{2^k}[x]/\left\langle x^2 - 1\right\rangle$. Since 2 is not invertible in $\mathbb{Z}_{2^k}[x]/\left\langle x^2 - 1\right\rangle$, we localize $\mathbb{Z}_{2^k}[x]/\left\langle x^2 - 1\right\rangle$ at the multiplicative set $2^{\mathbb{Z}_{\geq 0}}$ and find

$$\frac{2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}_{2^k}[x]}{\langle x^2 - 1\rangle}$$

the resulting localization. Consider the computation of Cooley–Tukey FFT mapping a polynomial $a_0 + a_1x$ to $(a_0 + a_1, a_0 - a_1)$. 2 is the largest power of two that we need to divide at the end of the computation and we only need to keep an additional bit for implementing the division by 2. Since initially we start with coefficients drawn from $\mathbb{Z}_{2^k}$, $\mathbb{Z}_{2^k} \to 2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}_{2^k} \to \mathbb{Z}_{2^{k+1}}$ is a suitable injection provided that divisions of 2 are issued only if the inputs are multiples of 2 as explained in previous paragraph. In summary, we compute with the following maps:

$$\frac{\mathbb{Z}_{2^k}[x]}{\langle x^2 - 1\rangle} \to \frac{\mathbb{Z}_{2^{k+1}}[x]}{\langle x^2 - 1\rangle} \to \frac{\mathbb{Z}_{2^{k+1}}[x]}{\langle x - 1\rangle} \times \frac{\mathbb{Z}_{2^{k+1}}[x]}{\langle x + 1\rangle}.$$

Notice that applying $\xi = z^{-k}\eta(\boldsymbol{a}) \mapsto z^{-1-k}\xi_z\left(\eta(\boldsymbol{a})\right)$ assumes an already existing approach for multiplying z^{-1} (bit-shift when $z = 2$). An alternative is to find ψ_{z_0} and ψ_{z_1} with $z_0 \perp z_1$ and integers e_0, e_1 satisfying $e_0z_0 + e_1z_1 = 1$, and define η^{-1} as

$$\eta^{-1} := r \mapsto e_0\psi_{z_0}(r) + e_1\psi_{z_1}(r).$$

Since e_0 and e_1 are integers, η^{-1} can be implemented entirely with arithmetic in R. [CK91] used localization and Schönhage's radix-2 and radix-3 FFTs [Sch77] for multiplying polynomials over arbitrary rings.

5.2 Schönhage's and Nussbaumer's Fast Fourier Transforms

Schönhage's and Nussbaumer's fast Fourier transforms adjoin special structures enabling fast monomorphisms. Schönhage's FFT extends the polynomial ring into a two-indeterminate polynomial ring while adjoining principal roots of unity, and splits the other dimension into smaller ones. Nussbaumer's FFT extends the polynomial ring into a two-indeterminate polynomial ring while adjoining a special shape of polynomial ring and splits it into smaller ones with the already existing principal roots of unity.

5.2.1 The General Cases

Let $\boldsymbol{g}(x^{n_1}) \in R[x]$ be a degree-$n_0 n_1$ monic polynomial. Schönhage's and Nussbaumer's FFTs exploit the structure of $\boldsymbol{g}(x^{n_1})$ by introducing $x^{n_1} \sim y$ and rewriting $R[x]/\langle \boldsymbol{g}(x^{n_1}) \rangle$ as $R[x,y]/\langle x^{n_1} - y, \boldsymbol{g}(y) \rangle$. We replace $x^{n_1} - y$ with $\boldsymbol{h}(x)$ satisfying $\deg(\boldsymbol{h}) \geq 2n_1 - 1$ and proceed differently for Schönhage's and Nussbaumer's FFTs.

Schönhage's FFT. Schönhage's FFT [Sch77] finds an invertible n satisfying $\boldsymbol{g}(y) | (y^n - 1)$ and $\boldsymbol{h}(x) | \Phi_n(x)$ in R and rewrites the polynomial ring as a polynomial ring over $R[x]/\langle \boldsymbol{h}(x) \rangle$:

$$\frac{R[x]}{\langle \boldsymbol{g}(x^{n_1}) \rangle} \cong \frac{R[x,y]}{\langle x^{n_1} - y, \boldsymbol{g}(y) \rangle} \hookrightarrow \frac{R[x,y]}{\langle \boldsymbol{h}(x), \boldsymbol{g}(y) \rangle} \cong \frac{\left(R[x]/\langle \boldsymbol{h}(x) \rangle \right)[y]}{\langle \boldsymbol{g}(y) \rangle}.$$

In the newly introduced polynomial ring $R[x]/\langle \boldsymbol{h}(x) \rangle$, we have $\Phi(x) = \boldsymbol{h}(x) \cdot \left(\Phi(x)/\boldsymbol{h}(x) \right) = 0 \cdot \left(\Phi(x)/\boldsymbol{h}(x) \right) = 0$ and x is a principal nth root of unity. Since $\boldsymbol{g}(x) | \left(y^n - 1 \right)$ over $\mathbb{Z}$ and $y^n - 1$ splits into $\prod_{i=0}^{n-1} \left(y - x^i \right)$ over $R[x]/\langle \boldsymbol{h}(x) \rangle$, $\boldsymbol{g}(x)$ splits into $\prod_{i \in \mathcal{I}} \left(y - x^i \right)$ over $R[x]/\langle \boldsymbol{h}(x) \rangle$ for an $\mathcal{I} \subset \{0, \dots, n-1\}$. This implies the following:

$$\frac{\left(R[x]/\langle \boldsymbol{h}(x) \rangle \right)[y]}{\langle \boldsymbol{g}(y) \rangle} \cong \prod_{i \in \mathcal{I}} \frac{\left(R[x]/\langle \boldsymbol{h}(x) \rangle \right)[y]}{\langle y - x^i \rangle}.$$

Nussbaumer's FFT. Nussbaumer's FFT [Nus80] finds an n satisfying $\boldsymbol{g}(y) | \Phi_n(y)$ over R and $\boldsymbol{h}(x) | (x^n - 1)$ in R and rewrites the polynomial ring as a polynomial ring over $R[y]/\langle \boldsymbol{g}(y) \rangle$:

$$\frac{R[x]}{\langle \boldsymbol{g}(x^{n_1}) \rangle} \cong \frac{R[x,y]}{\langle x^{n_1} - y, \boldsymbol{g}(y) \rangle} \hookrightarrow \frac{R[x,y]}{\langle \boldsymbol{h}(x), \boldsymbol{g}(y) \rangle} \cong \frac{\left(R[y]/\langle \boldsymbol{g}(y) \rangle \right)[x]}{\langle \boldsymbol{h}(x) \rangle}.$$

Similar to the arguments in Schönhage's FFT, y is a principal nth root of unity and $\boldsymbol{h}(x)$ splits into $\prod_{i \in \mathcal{I}} \left(x - y^i\right)$ for an $\mathcal{I} \subset \{0, \dots, n-1\}$. This implies the following:

$$\frac{\left(R[y]/\langle \boldsymbol{g}(y) \rangle\right)[x]}{\langle \boldsymbol{h}(x) \rangle} \cong \prod_{i \in \mathcal{I}} \frac{\left(R[y]/\langle \boldsymbol{g}(y) \rangle\right)[x]}{\langle x - y^i \rangle}.$$

See [MV83a, MV83b, Ber01] for more discussions generalizing the notion of principal roots of unity to automorphisms defining FFTs.

5.2.2 Radix-2 Cases

Cyclic Schönhage's FFT [Ber01, Section 9]. Cyclic Schönhage's FFT starts with the polynomial ring $R[x]\Big/\Big\langle x^{2^k} - 1 \Big\rangle$. We choose an $l \geq \frac{k}{2} - 1$, introduce the relation $x^{2^l} \sim y$, and replace the relation with $x^{2^{l+1}} \sim -1$. Define $\mathcal{R}' := R[x]\Big/\Big\langle x^{2^{l+1}} + 1 \Big\rangle$, and rewrite the polynomial ring as a polynomial ring with indeterminate y and coefficient ring $\mathcal{R}'$. Since $x^{2^{l+1}} = -1 \in \mathcal{R}'$ and $l + 2 \geq k - l$, $x^{2^{2l+2-k}}$ is a principal 2^{k-l}th root of unity defining a size-$(k-l)$ cyclic FFT. In summary, we have

$$\frac{R[x]}{\left\langle x^{2^k} - 1 \right\rangle} \cong \frac{R[x, y]}{\left\langle x^{2^l} - y, y^{2^{k-l}} - 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[y]}{\left\langle y^{2^{k-l}} - 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[y]}{\left\langle y - \boldsymbol{\omega}_{2^{k-l}}^i \right\rangle}$$

where $\boldsymbol{\omega}_{2^{k-l}} := x^{2^{2l+2-k}}$. The optimal choice is $l = \left\lceil \frac{k}{2} \right\rceil - 1$ leading to

$$\frac{R[x]}{\left\langle x^{2^k} - 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[y]}{\left\langle y^{2^{\lfloor k/2 \rfloor + 1}} - 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[y]}{\left\langle y - x^{2^{\lceil k/2 \rceil - \lfloor k/2 \rfloor \cdot l}} \right\rangle}$$

with $\mathcal{R}' = R[x]\Big/\Big\langle x^{2^{\lceil k/2 \rceil}} + 1 \Big\rangle$. Since multiplications by powers of x in $\mathcal{R}'$ amount to negacyclic shifts, we only need additions and subtractions for converting a polynomial multiplication in $R[x]\Big/\Big\langle x^{2^k} - 1 \Big\rangle$ into 2^{k-l} many polynomial multiplications in $R[x]\Big/\Big\langle x^{2^{l+1}} + 1 \Big\rangle$.

Negacyclic Schönhage's FFT [Sch77]. Negacyclic Schönhage's FFT starts with the polynomial ring $R[x]\Big/\Big\langle x^{2^k} + 1 \Big\rangle$. We choose an $l \geq \frac{k-1}{2}$ and proceed similarly in the cyclic case. This leads to

$$\frac{R[x]}{\left\langle x^{2^k} + 1 \right\rangle} \cong \frac{R[x, y]}{\left\langle x^{2^l} - y, y^{2^{k-l}} + 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[y]}{\left\langle y^{2^{k-l}} + 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[y]}{\left\langle y - \boldsymbol{\omega}_{2^{k-l+1}}^{1+2i} \right\rangle}$$

where $\mathcal{R}' := R[x] \Big/ \Big\langle x^{2^{l+1}} + 1 \Big\rangle$ and $\boldsymbol{\omega}_{2^{k-l+1}} := x^{2^{2l+1-k}}$. For the optimal choice $l = \lceil \frac{k-1}{2} \rceil$, we have

$$\frac{R[x]}{\left\langle x^{2^k} + 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[y]}{\left\langle y^{2^{\lfloor k+1/2 \rfloor}} + 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[y]}{\left\langle y - x^{2^{\lceil k-1/2 \rceil - \lfloor k-1/2 \rfloor} \cdot (1+2i)} \right\rangle}.$$

Nussbaumer's FFT [Nus80]. Nussbaumer's FFT is only applicable to the negacyclic case, but it sometimes results in smaller subproblems. We choose an $l \leq \frac{k}{2}$, introduce the relation $x^{2^l} \sim y$, and replace it with $x^{2^{l+1}} \sim 1$. We rewrite the polynomial ring as a polynomial ring with indeterminate x, and define $\mathcal{R}' := R[y] \Big/ \Big\langle y^{2^{k-l}} + 1 \Big\rangle$. Since $y^{2^{k-l}} = -1 \in \mathcal{R}'$, $y^{2^{k-2l}}$ is a principal 2^{l+1}th root of unity defining a size-2^{l+1} cyclic FFT. Overall, we have

$$\frac{R[x]}{\left\langle x^{2^k} + 1 \right\rangle} \cong \frac{R[x,y]}{\left\langle x^{2^l} - y, y^{2^{k-l}} + 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[x]}{\left\langle x^{2^{l+1}} - 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[x]}{\left\langle x - \boldsymbol{\omega}_{2^{l+1}}^i \right\rangle}$$

where $\boldsymbol{\omega}_{2^{l+1}} := y^{2^{k-2l}}$. For the optimal choice $l = \lfloor \frac{k}{2} \rfloor$, we have

$$\frac{R[x]}{\left\langle x^{2^k} + 1 \right\rangle} \hookrightarrow \frac{\mathcal{R}'[x]}{\left\langle x^{2^{\lfloor k/2 \rfloor + 1}} - 1 \right\rangle} \cong \prod_i \frac{\mathcal{R}'[x]}{\left\langle x - y^{2^{\lceil k/2 \rceil - \lfloor k/2 \rfloor} \cdot i} \right\rangle}.$$

Comparisons of Schönhage's and Nussbaumer's FFTs. Table 5.1 summarizes the domains, images, and defining conditions of radix-2 Schönhage's and Nussbaumer's FFTs, and Table 5.2 summarizes the domains and images of radix-2 Schönhage's and Nussbaumer's FFTs with optimal parameters. As seen in Table 5.2, for the negacyclic case $R[x] \Big/ \Big\langle x^{2^k} + 1 \Big\rangle$, Nussbaumer's FFT results in size-$\lceil \frac{k}{2} \rceil$ negacyclic convolutions and Schönhage's FFT results in size-$\lceil \frac{k+1}{2} \rceil$ negacyclic convolutions. This implies Nussbaumer's FFT is more preferable in the negacyclic case if the number of operations in R is the sole optimizing target [Ber01, Section 9].

Table 5.1: Overview of radix-2 Schönhage's and Nussbaumer's FFTs.

	Domain	Image	Condition
Cyclic Schönhage's FFT	$\frac{R[x]}{\left\langle x^{2^k}-1\right\rangle}$	$\left(\frac{R[x]}{\left\langle x^{2^{l+1}}+1\right\rangle}\right)^{2^{k-l}}$	$l \geq \frac{k}{2} - 1$
Negacyclic Schönhage's FFT	$\frac{R[x]}{\left\langle x^{2^k}+1\right\rangle}$	$\left(\frac{R[x]}{\left\langle x^{2^{l+1}}+1\right\rangle}\right)^{2^{k-l}}$	$l \geq \frac{k-1}{2}$
Nussbaumer's FFT	$\frac{R[x]}{\left\langle x^{2^k}+1\right\rangle}$	$\left(\frac{R[y]}{\left\langle y^{2^{k-l}}+1\right\rangle}\right)^{2^{l+1}}$	$l \leq \frac{k}{2}$

Table 5.2: Overview of optimal radix-2 Schönhage's and Nussbaumer's FFTs.

	Domain	Image
Cyclic Schönhage's FFT	$\frac{R[x]}{\left\langle x^{2^k}-1\right\rangle}$	$\left(\frac{R[x]}{\left\langle x^{2^{\lceil k/2\rceil+1}}+1\right\rangle}\right)^{2^{\lfloor k/2\rfloor+1}}$
Negacyclic Schönhage's FFT	$\frac{R[x]}{\left\langle x^{2^k}+1\right\rangle}$	$\left(\frac{R[x]}{\left\langle x^{2^{\lceil k+1/2\rceil}}+1\right\rangle}\right)^{2^{\lfloor k-1/2\rfloor}}$
Nussbaumer's FFT	$\frac{R[x]}{\left\langle x^{2^k}+1\right\rangle}$	$\left(\frac{R[y]}{\left\langle y^{2^{\lceil k/2\rceil}}+1\right\rangle}\right)^{2^{\lfloor k/2\rfloor+1}}$

Generalizations. There are several ways to generalize Schönhage's and Nussbaumer's FFTs. For the polynomial modulus $x^{2^k} \pm 1$ in Schönhage's FFT, the idea applies to any factors of $x^{2^k} \pm 1$. In fact, the case $x^{2^k} + 1$ directly follows from $x^{2^{k+1}} - 1$. See Section 6.3 for more discussions. Another direction is to replace x by an odd power of x.

5.3 Coefficient Ring Switching

For multiplying polynomials over $\mathbb{Z}_q[x]/\langle \boldsymbol{g} \rangle$ with $\boldsymbol{g}$ a polynomial in $\mathbb{Z}_q[x]$, we can always multiply in $\mathbb{Z}[x]/\langle \boldsymbol{g} \rangle$ and reduce to $\mathbb{Z}_q$ at the end. There are many ways to compute the result over $\mathbb{Z}$. Suppose we want to multiply two polynomials when $\boldsymbol{g} = x^n \pm 1$. For the signed representation of $\mathbb{Z}_q := \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$, since the result over $\mathbb{Z}$ has coefficients with absolute values bounded by $\frac{nq^2}{4}$, we choose a q' admitting a suitable FFT over $\boldsymbol{g}$ with $\frac{q'}{2} > \frac{nq^2}{4}$ and compute in $\mathbb{Z}_{q'}[x]/\langle \boldsymbol{g} \rangle$ with signed arithmetic. For unsigned arithmetic, the condition is replaced by $q' > nq^2$.

Applications. In many lattice-based cryptosystems, one of the operands has coefficients with absolute values bounded by a small constant, and q' only needs to be larger than a small-multiple of nq. For example, one of the operands in NTRU [CDH+20] has coefficients drawn from $\{0, \pm 1\}$ and the small secret vector of polynomials in Saber [DKRV20] has coefficients drawn from $[-3, 3] \cap \mathbb{Z}, [-4, 4] \cap \mathbb{Z}$, and $[-5, 5] \cap \mathbb{Z}$. Obviously, $\mathbb{Z}_q \hookrightarrow \mathbb{Z}_{q'}$ is a injective. If arithmetic defined over q' is too large for efficient implementations, one can also choose coprime integers q_i's and compute modulo all q_i's as long as their product $q' := \prod_i q_i$ fulfills the same conditions. The tuple of coprime integers is called a residue number system (RNS). Multiplying over $\mathbb{Z}_{q'}$ and $\prod_i \mathbb{Z}_{q_i}$ is used in many contexts, including lattice-based cryptosystems [FSS20, BBC+20, ACC+20, CHK+21, ACC+21], and also before asymmetric cryptography [Nic71, Pol71].

5.4 Comparisons

5.4.1 Localization, Schönhage's/Nussbaumer's FFT, Coefficient Ring Switching

We compare localization, Schönhage's/Nussbaumer's FFT, and coefficient ring switching in terms of the adjoined structures, bit-sizes of coefficient rings, and the degrees of polynomial moduli. See Table 5.3 for an overview.

Coefficient ring switching vs localization. Localization introduces inverses of integers, commonly 2^{-k}. In this case, we replace the coefficient ring with the k-bit larger one. Very often, we choose a k such that the new coefficient ring still amount to the same arithmetic precision, so there is usually no additional cost in practice. As for coefficient ring switching, since the bit-size is at least 2× larger, care must be taken while choosing the new coefficient ring.

Coefficient ring switching vs Schönhage's/Nussbaumer's FFT. Schönhage's and Nussbaumer's FFTs adjoin the principal roots of unity by extending the polynomial moduli, and result in $2\times$ number of coefficients. Coefficient ring switching introduces the principal roots by replacing the coefficient rings with much larger ones, and the polynomial moduli remain the same. To figure out which technique is more beneficial, programmers have to first figure out the efficiency of the multiplication in the coefficient rings. In Schönhage's and Nussbaumer's FFTs, the coefficient ring remains the same but we have doubly many elements. On the other hand, if we switch to a new coefficient ring, the bit-size of the new coefficient ring is at least $2\times$ larger than the original one. If the cost of multiplication in the original coefficient ring is very fast compared to the new large coefficient ring, then Schönhage's and Nussbaumer's FFTs might be more preferable.

Table 5.3: Overview of the cost of coefficient ring embedding.

Adjoined structure	Coeff. ring	Poly. modulus
Localization		
2^{-k}	k-bit larger	-
Schönhage's/Nussbaumer's FFT		
ω_{2^k}	-	$2\times$ #coeff.
Coeff. ring switching		
$2^{-k}, \omega_{2^k}$	$2^+\times$ larger	-

5.4.2 Schönhage's, Nussbaumer's, Cooley–Tukey FFTs

We compare the cost of Schönhage/Nussbaumer to Cooley–Tukey while multiplying two polynomials in $R[x]/\langle x^n+1\rangle$. For simplicity, we assume $n \geq 2$ is a power of two with exponent a power of two and there is a principal $2n$th root of unity in R. In the literature, a size-n Schönhage's/Nussbaumer's FFT for $R[x]/\langle x^n+1\rangle$ requires $\Theta\left(n \log_2 n \max\left(\log_2 \log_2 n, 1\right)\right)$ additions/subtractions and results in $\frac{n}{2}\log_2 n$ size-2 polynomials. If we multiply size-n polynomials with Schönhage's/Nussbaumer's FFT, we need $\Theta\left(n \log_2 n \max\left(\log_2 \log_2 n, 1\right)\right)$ operations in the coefficient ring. In practice, we need to revise the analysis of the number of multiplications for a concrete analysis. Suppose we recurse until the problem size is smaller than or equal to a platform-dependent power-of-two constant $t \geq 2$ with exponent a power of two and switch to asymptotically slower approaches, such as the schoolbook, Karatsuba, and Toom–Cook, with t^α operations where $1 < \alpha < 2$ is a constant. We revise the number of multiplications in Cooley–Tukey and Schönhage's/Nussbaumer's FFTs for polynomial multiplications as follows:

- Cooley–Tukey FFT: For the transformation, we need $\frac{n \log_2 n}{2 \log_2 t}$ multiplications for each, and there are three transformations, resulting in $\frac{3n \log_2 n}{2 \log_2 t}$ multiplications. Furthermore, we also have $\frac{n}{t}$ size-t polynomial multiplications with each requiring t^α multiplications. In total, we need $\frac{3n \log_2 n}{2 \log_2 t} + nt^{\alpha-1}$ multiplications with Cooley–Tukey FFT.

- Schönhage's/Nussbaumer's FFT: We do not need multiplications for the transformation, and we have $\frac{n \log_2 n}{t \log_2 t}$ size-t polynomial multiplications with each requiring t^α multiplications. Therefore, we need $\frac{nt^{\alpha-1} \log_2 n}{\log_2 t}$ multiplications with Schönhage's/Nussbaumer's FFT.

We compare the factors of the dominating term $n \log_2 n$: we have $\frac{3}{2 \log_2 t}$ in Cooley–Tukey and $\frac{t^{\alpha-1}}{\log_2 t}$ in Schönhage's/Nussbaumer's FFT. See Table 5.4 for a summary. Since t is typically between 4 to 16 by experiments [CHK+21, BBCT22], Schönhage's/Nussbaumer's FFT amounts to a much larger number of multiplications.

Table 5.4: Overview of the arithmetic cost of Schönhage's/Nussbaumer's and Cooley–Tukey FFTs for multiplying two size-n polynomials with the threshold t. We only give the number of multiplications for polynomial multiplication.

	Cooley-Tukey	Schönhage/Nussbaumer
Transformation		
# of mul.	$\frac{1}{2 \log_2 t} \cdot n \log_2 n$	0
# of add./sub.	$\frac{1}{\log_2 t} \cdot n \log_2 n$	$\Theta\left(n \log_2 n \max\left(\log_2 \log_t n, 1\right)\right)$
# of polymul.	$\frac{n}{t}$	$\frac{1}{t \log_2 t} \cdot n \log_2 n$
Polynomial multiplication		
# of mul.	$\frac{3n \log_2 n}{2 \log_2 t} + nt^{\alpha-1}$	$\frac{nt^{\alpha-1} \log_2 n}{\log_2 t}$

Chapter 6

The Choices of Polynomial Moduli

6.1 Embedding

For two size-n polynomials in $R[x]_{<n}$, we can compute their product with the **embedding** technique. We first identify a polynomial modulus $\boldsymbol{h}$ with degree larger than or equal to $2n-1$, and compute the product in the quotient ring $R[x]/\langle \boldsymbol{h} \rangle$. $\boldsymbol{h}$ is usually a polynomial with a very nice structure for fast transformations.

Evaluation at ∞ is an optimization for choosing $\boldsymbol{h}$ [Win80]. Suppose $\boldsymbol{r}$ is the product in $R[x]$, $d = 2n-1$ the maximum degree, and r_d the leading term of $\boldsymbol{r}$. Instead of computing $\boldsymbol{r}$, we compute $\boldsymbol{r} - r_d\boldsymbol{h}$ by embedding into $R[x]/\langle \boldsymbol{h} \rangle$ with $\deg(\boldsymbol{h}) = d$. The term $r_d\boldsymbol{h}$ is computed individually and added back. In the literature, the idea is commonly presented as allowing $\boldsymbol{h}$ to contain the polynomial $x - \infty$ as a factor. Historically, evaluation at ∞ was first used by [KO62]. [Too63] chose small integers for evaluation, and [Win80, Page 31] replaced a point with ∞ for unifying Karatsuba and Toom–Cook. [Win80]'s idea was already as general as this section and applied to other choices of $\boldsymbol{h}$.

Revisiting Karatsuba. In [KO62], they computed $(a_0 + a_1x)(b_0 + b_1x)$ with $(a_0+a_1x)(b_0+b_1x) = a_0b_0+((a_0+a_1)(b_0+b_1)-a_0b_0-a_1b_1)x+a_1b_1x^2$. If we choose $\boldsymbol{h} = x^2 + x$, the polynomial $(a_0 + a_1x)(b_0 + b_1x) - a_1b_1(x^2 + x) = a_0b_0 + (a_0b_1 + a_1b_0 - a_1b_1)x$ can be computed in $R[x]/\langle x^2 + x \rangle$. Applying $R[x]/\langle x^2 + x \rangle \cong R[x]/\langle x \rangle \times R[x]/\langle x - 1 \rangle$ gives us $(a_0, a_0 + a_1)$ and $(b_0, b_0 + b_1)$. After point-multiplying and inverting, we have $a_0b_0 + ((a_0 + a_1)(b_0 + b_1) - a_0a_1)x$. Adding $a_1b_1(x^2 + x)$ derives the desired result.

Revisiting the coefficient ring switching. Consider the coefficient ring switching in Section 5.3 while multiplying two size-n polynomials. We choose a new coefficient ring defining an efficient computation modulo a polynomial $\boldsymbol{g}$ with $\deg(\boldsymbol{g}) \geq 2n - 1$. Typically, $\deg(\boldsymbol{g})$ is smooth and preferably, $\deg(\boldsymbol{g})$ is a multiple of a high power of two. Since $2n - 1$ is odd, we have to search for an integer larger than $2n - 1$. With the embedding technique, we require $\deg(\boldsymbol{g}) \geq 2n-2$. If $2n-2$ happens to be a multiple of a high power of two, then we have a transformation with dimension smaller than before.

A small example for FFT along with evaluation at ∞. At the best of author's knowledge, $x - \infty$ is never chosen as a factor of $\boldsymbol{h}$ while applying FFT in the literature. The author believes the reason is that one usually splits $\boldsymbol{h}$ into a large number of small factors for FFT, and replacing one of them with $x - \infty$ results in marginal improvement. The following is example of multiplying $(a_0+a_1x)(b_0+b_1x)$ with evaluation at ∞ for referential purposes. We rewrite $(a_0 + a_1x)(b_0 + b_1x)$ as $(a_0b_0 + a_1b_1) + (a_0b_1 + a_1b_0)x + a_1b_1(x^2 - 1)$, compute $(a_0b_0 + a_1b_1) + (a_0b_1 + a_1b_0)x$ with the isomorphism $R[x]/\langle x^2 - 1\rangle \cong \prod R[x]/\langle x \pm 1\rangle$, and finally add $a_1b_1(x^2 - 1)$ to the result.

6.2 Twisting

Let $\zeta \in R$ be an invertible element. **Twisting** is an isomorphism from $R[x]/\langle \boldsymbol{g}(x)\rangle$ to $R[y]/\langle \boldsymbol{g}(\zeta y)\rangle$ by introducing $x \sim \zeta y$. We have the isomorphism $R[x]/\langle \boldsymbol{g}(x)\rangle \cong R[x, y]/\langle x - \zeta y, \boldsymbol{g}(\zeta y)\rangle$ and treat $R[x]/\langle x - \zeta y\rangle$ as the coefficient ring. Let $n = \deg(\boldsymbol{g})$. While changing the basis from $(1, x, \dots, x^{n-1})$ to $(1, y, \dots, y^{n-1}) = (1, \zeta x, \dots, \zeta^{n-1}x^{n-1})$, we have to multiply the coefficients with the powers $\zeta, \dots, \zeta^{n-1}$. This usually amounts to $n - 1$ multiplications in R. However, if n is odd and $\zeta = -1$, we do not need any multiplication for the isomorphism $R[x]/\langle x^n + 1\rangle \cong R[x, y]/\langle x + y, y^n - 1\rangle$. We will approach this observation in a systematic manner in Section 6.3

[GS66] introduced twisting while computing FFTs with $R[x]/\langle x^{n_0n_1} - 1\rangle \cong \prod_i R[x]/\langle x^{n_1} - \omega_{n_0}^i\rangle \cong \prod_i R[x]/\langle x^{n_1} - 1\rangle$. See [DH84, Für09] for more insights on the choices of n_0 and n_1.

Composed multiplication. **Composed multiplication** is a specialized approach when $R = \mathbb{F}_q$. Given $\boldsymbol{f}_0, \boldsymbol{f}_1 \in \mathbb{F}_q[x]$, we defined their composed multiplication [BC87] as

$$\boldsymbol{f}_0 \odot \boldsymbol{f}_1 \coloneqq \prod_{\boldsymbol{f}_0(\alpha)=0} \prod_{\boldsymbol{f}_1(\beta)=0} (x - \alpha\beta)$$

where α, β are elements from an extension field of $\mathbb{F}_q$ containing the splitting fields of $\boldsymbol{f}_0, \boldsymbol{f}_1$. Composed multiplication generalizes twisting to the polynomial modulus of the form $(x-\zeta)\odot\boldsymbol{f}(x)$: $\mathbb{F}_q[x]/\langle (x-\zeta)\odot \boldsymbol{f}(x)\rangle \cong \mathbb{F}_q[y]/\langle x - \zeta y, \boldsymbol{f}(y)\rangle$.

Factorizing with composed multiplication. Another benefit of composed multiplication is to systematically derive transformations from presumably simpler coprime factorizations. Let $\boldsymbol{f}_0 = \prod_{i_0} \boldsymbol{f}_{0,i_0}$ and $\boldsymbol{f}_1 = \prod_{i_1} \boldsymbol{f}_{1,i_1}$ be coprime factorizations in $\mathbb{F}_q[x]$. We have $\boldsymbol{f}_0 \odot \boldsymbol{f}_1 = \prod_{i_0} \left(\boldsymbol{f}_{0,i_0} \odot \boldsymbol{f}_1\right) = \prod_{i_0,i_1} \left(\boldsymbol{f}_{0,i_0} \odot \boldsymbol{f}_{1,i_1}\right)$. A practically important example is $\boldsymbol{f}_0 = x^r - 1 = \prod_{i_0} \left(x - \omega_r^{i_0}\right) \in \mathbb{F}_q[x]$ and $\boldsymbol{f}_1 = x^{2^k} - 1$. Given a factorization $x^{2^k} - 1 = \prod_{i_1} \boldsymbol{f}_{1,i_1}$ in $\mathbb{F}_q[x]$, we have

$$x^{2^k r} - 1 = \prod_{i_0} \left(x^{2^k} - \omega_r^{2^k i_0}\right) = \prod_{i_0,i_1} \omega_r^{i_0 \deg(\boldsymbol{f}_{1,i_1})} \boldsymbol{f}_{1,i_1}(\omega_r^{-i_0} x).$$

A small example. Consider $x^3 - 1 = (x-1)(x-\omega_3)\left(x - \omega_3^2\right)$ and $x^8 - 1 = \left(x^4 - 1\right)\left(x^2 - \sqrt{2}x + 1\right)\left(x^2 + \sqrt{2}x + 1\right) \in \mathbb{F}[x]$ for $\omega_3 \in \mathbb{F}$ an element satisfying $1 + \omega_3 + \omega_3^2 = 0 \in \mathbb{F}$ and $\sqrt{2} \in \mathbb{F}$ an element whose square equals to 2. We split $x^{24} - 1$ as follows:

$$\begin{aligned} x^{24} - 1 = \quad & \left(x^8 - 1\right)\left(x^8 - \omega_3^2\right)\left(x^8 - \omega_3\right) \\ = \quad & \left(x^4 - 1\right)\left(x^2 - \sqrt{2}x + 1\right)\left(x^2 + \sqrt{2}x + 1\right) \\ & \left(x^4 - \omega_3\right)\left(x^2 - \sqrt{2}\omega_3 x + \omega_3^2\right)\left(x^2 + \sqrt{2}\omega_3 x + \omega_3^2\right) \\ & \left(x^4 - \omega_3^2\right)\left(x^2 - \sqrt{2}\omega_3^2 x + \omega_3\right)\left(x^2 + \sqrt{2}\omega_3^2 x + \omega_3\right). \end{aligned}$$

As $x^4 - 1 = (x-1)(x+1)(x^2+1) \in \mathbb{F}[x]$, we can split $x^4 - \omega_3$ and $x^4 - \omega_3^2$ similarly.

6.3 Truncation

Truncation is a simple and powerful idea: Let $\mathcal{I}' \subset \mathcal{I}$ be index sets and $\{\boldsymbol{g}_i\}_{i\in\mathcal{I}}$ be coprime polynomials in $R[x]$. Suppose we are given the following isomorphism

$$\eta : \begin{cases} R[x] \Big/ \left\langle \prod_{i \in \mathcal{I}} \boldsymbol{g}_i \right\rangle & \to \quad \prod_{i \in \mathcal{I}} R[x]/\langle \boldsymbol{g}_i \rangle \ , \\ \boldsymbol{a} & \mapsto \quad (\boldsymbol{a} \bmod \boldsymbol{g}_i)_{i \in \mathcal{I}} \ . \end{cases}$$

We can naturally define an isomorphism $\eta_{\mathcal{I}'}$ as

$$\eta_{\mathcal{I}'} : \begin{cases} R[x] \Big/ \left\langle \prod_{i \in \mathcal{I}'} \boldsymbol{g}_i \right\rangle & \to \quad \prod_{i \in \mathcal{I}'} R[x]/\langle \boldsymbol{g}_i \rangle \ , \\ \boldsymbol{a} & \mapsto \quad (\boldsymbol{a} \bmod \boldsymbol{g}_i)_{i \in \mathcal{I}'} \ . \end{cases}$$

$\eta_{\mathcal{I}'}$ is called the **truncation of** η **at** $R[x]/\langle \prod_{i \in \mathcal{I}'} \boldsymbol{g}_i \rangle$. Truncation was introduced by [CF94, Section 7]. [Ber08b] described the benefit in terms of complexity (according to [vdH04], the work [Ber08b] was already online prior to [vdH04]), and [vdH04] named the technique "truncated Fourier transform" for the FFT case. We call it truncation since it is not restricted to FFTs.

6.3.1 Application I: Negacyclic From Cyclic

We derive FFT for $R[x]\Big/\left\langle x^{2^{k-1}} + 1 \right\rangle$ from the one for $R[x]\Big/\left\langle x^{2^k} - 1 \right\rangle$. For a principal 2^kth root of unity ω_{2^k} implementing

$$R[x]\Big/\left\langle x^{2^k} - 1 \right\rangle \cong \prod_{i=0}^{2^k - 1} R[x]\Big/\left\langle x - \omega_{2^k}^i \right\rangle \ ,$$

we have the isomorphism

$$R[x]\Big/\left\langle x^{2^{k-1}} + 1 \right\rangle \cong \prod_{i=0}^{2^{k-1} - 1} R[x]\Big/\left\langle x - \omega_{2^k}^{1+2i} \right\rangle$$

as $x^{2^{k-1}} + 1 = \prod_{i=0}^{2^{k-1}-1} \left(x - \omega_{2^k}^{1+2i}\right)$ is a factor of $x^{2^k} - 1 = \prod_{i=0}^{2^k - 1} \left(x - \omega_{2^k}^i\right)$. We generalize the idea to arbitrary transformation sizes. Below is a straightforward generalization of [CF94, Section 7] outlined in [Hwa22, Section 10]. Let $b = n$ and $\tilde{b} = \sum_j \tilde{b}_j 2^j$ be the 2's complement representation of $-n$ as a $\lceil \log_2 n \rceil$-bit integer. We have $b + \tilde{b} = 2^{\lceil \log_2 n \rceil}$ by definition and define a transformation for

$$R[x]\Bigg/\left\langle \frac{x^{2^{\lceil \log_2 n \rceil}} - 1}{\prod_j \left(x^{2^j} + 1\right)^{\tilde{b}_j}} \right\rangle .$$

This boils down to transformations for rings of the form $R[x]\Big/\Big\langle x^{2^k} \pm 1\Big\rangle$. An example is the Schönhage for $R[x]/\langle (x^{1024}+1)(x^{512}-1)\rangle$ derived from $R[x]/\langle x^{2048}-1\rangle$. See [MV83b] for more explanations on the choices of polynomial moduli.

6.3.2 Application II: Rader's FFT

Let p be an odd prime, $\mathcal{I} = \{0, \dots, p-1\}$, $\mathcal{I}^* = \{z \in \mathcal{I} | z \perp p\}$, and g be a generator of $\mathcal{I}^*$. For a principal pth root of unity, Rader's FFT (cf. Section 4.7) converts the computing task of size-p cyclic FFT into a size-$\lambda(p)$ cyclic convolution. In this section, we show that the isomorphism $R[x]/\langle \prod_{i\in\mathcal{I}^*} (x-\omega_p^i)\rangle \cong \prod_{i\in\mathcal{I}^*} R[x]/\langle x-\omega_p^i\rangle$ and its inverse can also be converted into size-$\lambda(p)$ cyclic convolutions. For generalization truncating a size-n cyclic DFT to the roots with exponents coprime to n, see [Ber23, Sections 4.12.3 and 4.12.4].

Forward transformation. For $\sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j x^j \in R[x]/\langle \prod_{i\in\mathcal{I}^*} (x-\omega_p^i)\rangle$ and its image $(\hat{a}_{i-1})_{i\in\mathcal{I}^*} = \sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j x^j \bmod \left(x-\omega_p^i\right)$, we have:

$$\begin{aligned} \hat{a}_{g^{\log_g i}-1} = \hat{a}_{i-1} = \sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j \omega_p^{ij} = \omega_p^{-i} \sum_{j\in\mathbb{Z}_{\lambda(p)}} a_j \omega_p^{i(j+1)} = \omega_p^{-i} \sum_{j\in\mathcal{I}^*} a_{j-1}\omega_p^{ij} \\ = \quad \omega_p^{-g^{\log_g i}} \sum_{-\log_g j\in\mathbb{Z}_{\lambda(p)}} a_{g^{\log_g j}-1}\omega_p^{g^{\log_g i+\log_g j}}. \end{aligned}$$

If we multiply both sides by $\omega_p^{g^{\log_g i}}$, then we find that $\left(\omega_p^{g^k}\hat{a}_{g^k-1}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is a size-$\lambda(p)$ cyclic convolution of $\left(a_{g^{-k}-1}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_n^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$.

Inverse transformation. As polynomial multiplication in $R[x]\Big/\Big\langle x^{\lambda(p)}-1\Big\rangle$ is the ring multiplication in the group algebra $R[\mathbb{Z}_{\lambda(p)}]$ (cf. Section 2.2), the inversion of the transformation amounts to multiplying the multiplicative inverse of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ in the group algebra $R[\mathbb{Z}_{\lambda(p)}]$. The inverse of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is $\frac{1}{p}\left(\omega_n^{-g^{-k}}-1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$. [Ber23, Section 4.8.2] proved this by showing that the convolution of $\left(\omega_p^{g^k}\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ and $\left(\omega_p^{-g^{-k}}-1\right)_{k\in\mathbb{Z}_{\lambda(p)}}$ is $(\delta_{0,k}p)_{k\in\mathbb{Z}_{\lambda(p)}}$: For all $k \in \mathbb{Z}_{\lambda(p)}$, we find

$$\sum_{i+j=k} \omega_n^{g^i}\left(\omega_n^{-g^{-j}}-1\right) = \sum_{i+j=k} \omega_n^{g^i\left(1-g^{-(i+j)}\right)} - \sum_{i+j=k} \omega_n^{g^i} = \delta_{0,k}p.$$

6.4 Incomplete Transformation and Striding

6.4.1 Incomplete Transformation

For a monic polynomial $\boldsymbol{g}(x^v) \in R[x]$, a homomorphism $f : R[x]/\langle \boldsymbol{g}(x^v) \rangle \to \mathcal{A}$ is called **incomplete** if f starts with introducing $x^v \sim y$ and proceed as a polynomial ring in y with the coefficient ring $R[x]/\langle x^v - y \rangle$. There are several benefits for an incomplete transformation: (i) the definability of fast transformation, (ii) the vectorization-friendliness of $x^v \sim y$, and (iii) the code size for implementing f. We give an example for (i) in this section. As for (iii), we refer to [AHY22, Sections 3.2 and 3.3] for more details.

Real-world example(s). Take the polynomial ring $\mathbb{Z}_{3329}[x]/\left\langle x^{256} + 1 \right\rangle$ used in Kyber as an example. Since 3329 is a prime, we can only define a size-n cyclic FFT for $n|3328$. This does not permit splitting the polynomial ring into linear factors since $x^{256} + 1 = \Phi_{512}$ and 512 is not a factor of 3328. What we can do is to introduce $x^2 \sim y$ and split $\left(\mathbb{Z}_{3329}[x]/\left\langle x^2 - y \right\rangle \right)[y]/\left\langle y^{128} + 1 \right\rangle$ into linear factors in y.

6.4.2 Striding

A closely related idea is **striding** – we regard $R[y]/\langle \boldsymbol{g}(y) \rangle$ as the coefficient ring. This is Nussbaumer if we replace $x^v - y$ with an $\boldsymbol{h}(x)$, and ask $\boldsymbol{g}(y)|\Phi_{n'}(y)$ and $\boldsymbol{h}(x)|(x^{n'} - 1)$ with $n' \geq 2v - 1$. We also have striding Toom–Cook [Ber01, BMK+21] if $\boldsymbol{h}(x) = \prod_i (x - s_i)$ for $\{s_i\} \subset \mathbb{Q} \cup \{\infty\}$.

Real-world example(s). Consider the polynomial ring $\mathbb{Z}_{8192}[x]/\left\langle x^{256} + 1 \right\rangle$. We rewrite the polynomial ring as

$$\frac{\mathbb{Z}_{8192}[x]}{\langle x^{256} + 1 \rangle} \cong \frac{\left(\mathbb{Z}_{8192}[y]/\left\langle y^{64} + 1 \right\rangle \right)[x]}{\langle x^4 - y \rangle},$$

and apply Toom-4 in x.

6.4.3 Comparisons

We compare incomplete transformation and striding, and illustrate their applications with coefficient switching in Section 5.3 and embedding in Section 6.1. Suppose we want to multiply polynomials in the polynomial ring $R[x]/\langle \boldsymbol{g}(x^v) \rangle$. If $R[x]/\langle \boldsymbol{g}(x^v) \rangle$ does not split at all, we have two choices:

- Replace the coefficient ring R with a larger ring R' such that the result in $R[x]/\langle \boldsymbol{g}(x^v)\rangle$ can be recovered from $R'[x]/\langle \boldsymbol{g}(x^v)\rangle$ and $R'[x]/\langle \boldsymbol{g}(x^v)\rangle$ splits into small-dimensional polynomial rings with an incomplete transformation.
- Rewrite the polynomial ring as $R[x]/\langle \boldsymbol{g}(x^v)\rangle \cong \left(R[y]/\langle \boldsymbol{g}(y)\rangle\right)[x]/\langle x^v - y\rangle$ with striding and embed it to a polynomial ring modulo a polynomial with degree larger than or equal to $2v - 1$.

Figure 6.1 is an overview of the two approaches.

Figure 6.1: Overview of approaches built upon coefficient ring switching, embedding, incomplete transformation, and striding.

$$\begin{tikzcd}
 & \frac{R[x]}{\langle \boldsymbol{g}(x^v)\rangle} \arrow[dl, "\text{Striding}"'] \arrow[dr, "\text{Coefficient ring switching}"] & \\
\frac{\frac{R[y]}{\langle \boldsymbol{g}(y)\rangle}[x]}{\langle x^v - y\rangle} \arrow[d, "\text{Embedding}"'] & & \frac{R'[x]}{\langle \boldsymbol{g}(x^v)\rangle} \arrow[d, "\text{Incomplete trans.}"] \\
\prod_j \frac{\frac{R[y]}{\langle \boldsymbol{g}(y)\rangle}[x]}{\langle \boldsymbol{h}_j(x)\rangle} & & \prod_i \frac{\frac{R'[x]}{\langle x^v - y\rangle}[y]}{\langle \boldsymbol{g}_i(y)\rangle}
\end{tikzcd}$$

6.5 Toeplitz Matrix-Vector Product

This section goes through a generic technique converting a fast computation for $R[x]$ into a Toeplitz-matrix-vector-product-based fast computation for $R[x]/\langle x^n - \alpha x - \beta\rangle$.

6.5.1 Bilinear Maps

Bilinear maps. Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be modules over the ring R. We call a map $\eta : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$ a **bilinear map** if

- $\forall \boldsymbol{a} \in \mathcal{A}, \eta(\boldsymbol{a}, -) : \mathcal{B} \to \mathcal{C}$ is a module homomorphism, and
- $\forall \boldsymbol{b} \in \mathcal{B}, \eta(-, \boldsymbol{b}) : \mathcal{A} \to \mathcal{C}$ is a module homomorphism.

Suppose we have maps $\psi : \mathcal{A}^* \to \mathcal{A}', \kappa : \mathcal{B} \to \mathcal{B}', \iota : \mathcal{C}' \to \mathcal{C}^*$, and a bilinear map $\xi : \mathcal{C}' \times \mathcal{B}' \to \mathcal{A}'$ satisfying

$$\forall \boldsymbol{b} \in \mathcal{B}, \xi\left(-, \kappa\left(\boldsymbol{b}\right)\right) = \psi \circ \eta\left(-, \boldsymbol{b}\right)^* \circ \iota.$$

If $\eta\left(-, \boldsymbol{b}\right) = f_{\boldsymbol{b}} \circ g_{\boldsymbol{b}}$ for some $f_{\boldsymbol{b}}$ and $g_{\boldsymbol{b}}$, we have the corresponding factorization for $\xi\left(-, \kappa\left(\boldsymbol{b}\right)\right)$:

$$\forall \boldsymbol{b} \in \mathcal{B}, \xi\left(-, \kappa\left(\boldsymbol{b}\right)\right) = \psi \circ g_{\boldsymbol{b}}^* \circ f_{\boldsymbol{b}}^* \circ \iota.$$

Assume $\mathcal{A}' = \mathcal{A}, \mathcal{B}' = \mathcal{B}$, $\mathcal{C}' = \mathcal{C}$, $\psi = \boldsymbol{a}^* \mapsto \boldsymbol{a}$, $\kappa = \mathrm{id}_{\mathcal{B}}$, and $\iota = \boldsymbol{c} \mapsto \boldsymbol{c}^*$. For finite index sets $\mathcal{I}, \mathcal{J}, \mathcal{K}$ and $(r_{(i,j,k)})_{(i,j,k) \in \mathcal{I} \times \mathcal{J} \times \mathcal{K}}$, define $\boldsymbol{a} = (a_i)_{i \in \mathcal{I}} \in \mathcal{A}, \boldsymbol{b} = (b_j)_{j \in \mathcal{J}} \in \mathcal{B}, \boldsymbol{c} = (c_k)_{k \in \mathcal{K}} \in \mathcal{C}$. Then, we write

$$\begin{cases} \left(\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} r_{(i,j,k)} a_i b_j\right)_{k \in \mathcal{K}} & = \eta(-, \boldsymbol{b})(\boldsymbol{a}), \\ \left(\sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}} r_{(i,j,k)} c_k b_j\right)_{i \in \mathcal{I}} & = \xi(-, \boldsymbol{b})(\boldsymbol{c}), \end{cases}$$

and find $\xi(-, \boldsymbol{b}) = (\psi \circ \eta(-, \boldsymbol{b})^* \circ \iota)$.

Bilinear maps, concretely. We review a generic technique for bilinear maps adapted from [Win80, Theorem 6].

Theorem 7 ([Win80, Theorem 6] for R commutative)**.** Let R be a ring, $\mathcal{I}, \mathcal{J}, \mathcal{K}$ be finite index sets, and $(a_i)_{i \in \mathcal{I}}, (b_j)_{j \in \mathcal{J}}, (c_k)_{k \in \mathcal{K}}$ be tuples drawn from R. For a bilinear system

$$S_0 : \forall k \in \mathcal{K}, \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} r_{(i,j,k)} a_i b_j$$

with $r_{(i,j,k)} \in R$, we construct the following bilinear systems:

$$S_1 : \forall j \in \mathcal{J}, \sum_{i \in \mathcal{I}} \sum_{k \in \mathcal{K}} r_{(i,j,k)} a_i c_k,$$

$$S_2 : \forall i \in \mathcal{I}, \sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}} r_{(i,j,k)} c_k b_j.$$

Then any bilinear algorithm for one of S_0, S_1 or S_2 leads to algorithms for the other two.

For proving Theorem 7, we define a $|\mathcal{K}| \times |\mathcal{I}|$ matrix $B_{k,i} := \left(\sum_{j \in \mathcal{J}} r_{(i,j,k)} b_j\right)$, and write S_0 as $B\boldsymbol{a}$ and S_2 as $B^T\boldsymbol{c}$ where $\boldsymbol{a}$ and $\boldsymbol{c}$ are the column representations of $\left(a_i\right)_{i \in \mathcal{I}}$ and $\left(c_k\right)_{k \in \mathcal{K}}$. If we choose $r_{(i,j,k)} := [\![i + j = k]\!]$ where $[\![ ]\!]$ is the **Iverson bracket**[1] and $|\mathcal{K}| = |\mathcal{I}| + |\mathcal{J}| - 1$, S_0 represents the coefficients of $\left(\sum_{i\in\mathcal{I}} a_i x^i\right)\left(\sum_{j\in\mathcal{J}} b_j x^j\right)$ in an obvious way. Then, S_2 becomes

$$S_2 : \forall i \in \mathcal{I}, \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}} [\![k - j = i]\!]\, c_k b_j .$$

This is called a **transposed multiplication** [Sho99, Section 3] or a **middle product** [HQZ04]. [Fid73, Theorem 4 and 5] proved the **transposition principle**: transposing an algorithm results in same numbers of constant multiplications (ra_i for a constant r in R), non-constant multiplications ($a_i b_j$), and additions/subtractions with a linear difference. See [BCS13, Section 4] for the history of transposition principle.

We illustrate with the case $|\mathcal{I}| = |\mathcal{J}| = n$. For the bilinear system $S_0 : \forall k \in \mathcal{K}, \sum_{i\in\mathcal{I}}\sum_{j\in\mathcal{J}} [\![i + j = k]\!]\, a_i b_j = \sum_{i\in\mathcal{I}, i\le k} a_i b_{k-i}$, we write it as follows:

$$\begin{pmatrix} a_0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots \\ a_{n-1} & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix} .$$

And similarly for the bilinear system $S_2 : \forall i \in \mathcal{I}, \sum_{j\in\mathcal{J}}\sum_{k\in\mathcal{K}} [\![k - j = i]\!]\, c_k b_j = \sum_{j\in\mathcal{J}} c_{i+j} b_j$:

$$\begin{pmatrix} c_0 & \iddots & \iddots \\ \vdots & \iddots & \iddots \\ c_{n-1} & \cdots & c_{2n-2} \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix} .$$

S_2 relates S_0 to polynomial multiplication modulo a polynomial.

6.5.2 Toeplitz Transformation for $R[x]/\langle x^n - \alpha x - \beta\rangle$

Let M be an $n \times n$ matrix. We call M a **Hankel matrix** if $M_{i,j} = M_{i+1,j-1}$ for all possible i, j, and a **Toeplitz matrix** if $M_{i,j} = M_{i+1,j+1}$ for all possible i, j. Notice that a Hankel matrix can be converted into a Toeplitz matrix by multiplying an anti-diagonal matrix of ones and vice versa.

---

[1] Iverson bracket is an indicator function for the truthfulness. The image of $[\![]\!]$ is $\{0, 1\}$, which can be certainly embedded into a ring.

This section explains how to derive a Toeplitz-matrix-vector-product-based fast computation for $R[x]/\langle x^n - \alpha x - \beta \rangle$ from an already well-studied algebra homomorphism f multiplying two size-n polynomials in $R[x]$. There are four steps: (i) interpreting multiplication in $R[x]/\langle x^n - \alpha x - \beta \rangle$ as a Toeplitz matrix-vector product; (ii) interpreting the Toeplitz matrix-vector product as a composition of applying an anti-diagonal matrix of ones and a Hankel matrix–vector product; (iii) rewriting the Hankel matrix–vector product as a bilinear system of the form S_2; and (iv) converting the computing task into a bilinear system of the form S_0. Once we go through all the steps (i) – (iv), we can now convert an f into an algorithm for $R[x]/\langle x^n - \alpha x - \beta \rangle$ via the module–theoretic view. Notice that steps (ii) and (iii) are sometimes described as a single step. We describe them separately for clarity.

Steps (i) – (iii) are already shown in previous paragraphs. We now explain how to interpret the multiplication in $R[x]/\langle x^n - \alpha x - \beta \rangle$ as a Toeplitz matrix-vector product with potential post-processing. We define $\mathbf{Toeplitz}_n$ as the following function mapping a $(2n-1)$-tuple drawn from R to a Toeplitz matrix over R:

$$\mathbf{Toeplitz}_n : (z_{2n-2}, \dots, z_0) \mapsto \begin{pmatrix} z_{n-1} & \cdots & z_0 \\ \vdots & \ddots & \ddots \\ z_{2n-2} & \ddots & \ddots \end{pmatrix}.$$

Let $\boldsymbol{a} = \sum_i a_i x^i, \boldsymbol{b} = \sum b_i x^i$ be size-n polynomials. We recall that computing $\sum_i c_i x^i = \boldsymbol{ab}$ in $R[x]$ can be regarded as the following matrix–vector product:

$$\begin{pmatrix} c_0 \\ \vdots \\ c_{2n-2} \end{pmatrix} = \begin{pmatrix} b_0 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots \\ b_{n-1} & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Since $(c_0, \dots, c_{n-1})$ can be computed with a Toeplitz matrix-vector product, we only need to convert reduction modulo $x^n - \alpha x - \beta$ into the manipulation of Toeplitz matrices. A standard approach for reducing modulo $x^n - \alpha x - \beta$ is multiplying $(c_n, \dots, c_{2n-2})$ by α and β and adding the results to $(c_1, \dots, c_{n-1})$ and $(c_0, \dots, c_{n-2})$. Based on this, $\boldsymbol{ab} \bmod (x^n - \alpha x - \beta)$ can be written as

$$(M_0 + M_1 + M_2)\, \boldsymbol{a}$$

where

$$\begin{cases} M_0 = \mathbf{Toeplitz}_n(b_{n-1}, \dots, b_0, 0, \dots, 0), \\ M_1 = \mathbf{Toeplitz}_n(0, \dots, 0, \beta b_{n-1}, \dots, \beta b_1), \end{cases}$$

and

$$M_2 = \alpha \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 0 & b_{n-1} & \cdots & b_0 \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \ddots & \ddots & \ddots \end{pmatrix}.$$

Observe $M_2 = M_2' - \begin{pmatrix} \alpha b_{n-1} & \cdots & \alpha b_1 & 0 \\ 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \end{pmatrix}$ for

$$M_2' = \mathbf{Toeplitz}_n(0, \dots, 0, \alpha b_{n-1}, \dots, \alpha b_1, 0),$$

we rewrite $\boldsymbol{ab} \bmod (x^n - \alpha x - \beta)$ as follows

$$\boldsymbol{ab} \bmod (x^n - \alpha x - \beta) = \left(M_0 + M_1 + M_2'\right) \boldsymbol{a} - \begin{pmatrix} \alpha b_{n-1} & \cdots & \alpha b_1 & 0 \\ 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 \end{pmatrix} \boldsymbol{a}$$

where $M_0 + M_1 + M_2'$ is the following Toeplitz matrix

$$\mathbf{Toeplitz}_n(b_{n-1}, \dots, b_1, b_0 + \alpha b_{n-1}, \beta b_{n-1} + \alpha b_{n-2}, \dots, \beta b_2 + \alpha b_1, \beta b_1) \text{ [Yan23].}$$

A specialized approach for $\beta = 1$. We review the case $\beta = 1$ implied by [FH07, Section 3.2]. See [HB95, FD05] for related works when $R = \mathbb{F}_2$. Since $\beta = 1$, $M_0 + M_1$ is the circulant matrix implementing $\boldsymbol{ab} \bmod (x^n - 1)$. Obviously, if we multiply a circulant matrix by a cyclic shift matrix (either left-multiplying or right-multiplying), we still have a circulant matrix. Let P be the matrix moving the 0th row of a circulant matrix to the bottom. We find that both $P(M_0 + M_1)$ and PM_2 are Toeplitz matrices. Therefore, $P(M_0 + M_1 + M_2)$ is a Toeplitz matrix and we can implement $(M_0 + M_1 + M_2)\,\boldsymbol{a}$ as

$$(M_0 + M_1 + M_2)\,\boldsymbol{a} = P^{-1}\left(P\left(M_0 + M_1 + M_2\right)\boldsymbol{a}\right).$$

Padding. The last instrument is padding. Suppose we have an $n \times n$ Toeplitz matrix $T = \textbf{Toeplitz}\left(z_{2n-2}, \dots, z_0\right)$. For an $n' \geq n$, we can always pad T to an $n' \times n'$ Toeplitz matrix T' as follows:

$$T' = \textbf{Toeplitz}\left(\underbrace{0, \dots, 0}_{n'-n}, z_{2n-2}, \dots, z_0, \underbrace{0, \dots, 0}_{n'-n}\right).$$

The point is that if a $n \times n$ Toeplitz matrix does not admit efficient implementations, we can pad them to slightly larger ones with efficient implementations [IKPC22, Section 3.1].

Chapter 7

Vectorization

This chapter goes through the formalization of vectorization and its relation to the design of fast homomorphisms.

7.1 Vectorization-Friendliness

This section reviews "vectorization-friendliness" formally relating homomorphisms to vector-by-vector instructions [Hwa24c]. Conceptually, vectorization-friendliness qualifies if a homomorphism can be mapped to a string of vector-by-vector instructions and cyclic/negacyclic shifts. Cyclic and negacyclic shifts are vectorization-friendly since we can implement them with extractions or memory operations:

- Extractions: For cyclic shift, we extract consecutive elements from a pair of SIMD registers and extract again with inputs swapped. The resulting pair of SIMD registers is now a cyclic shift of the original pair. For the negacyclic shift, we replace a SIMD register by its negative value in one of the extractions. This idea is applicable to Armv7/8-A since we have `ext` implementing exactly the desired operations [HLY24].

- Memory operations: We can also implement cyclic/negacyclic shift with memory operations – we perform unaligned loads, shuffle the last SIMD register (and negate it in the negacyclic case), and store the vectors to memory [BBCT22].

The set `BlockDiag`. We define `BlockDiag` as a certain set of block diagonal matrices implementing cyclic/negacyclic shifts and twisting. Formally, `BlockDiag` is defined as a set of all possible block diagonal matrices with each block a $v' \times v'$ matrix that is a diagonal matrix implementing twisting or a cyclic/negacyclic shift matrix for all v-multiple v'.

Vectorization-friendliness, formally [Hwa24c]. Let f be an algebra homomorphism and M_f its matrix form. f is called **vectorization-friendly** if

$$M_f = \prod_i \left(M_{f_i} \otimes I_v \right) S_{f_i}$$

for $S_{f_i} \in$ `BlockDiag` and some matrices M_{f_i}s. Once we find such a decomposition for a vectorization-friendly f, we implement $M_{f_i} \otimes I_v$ with vector additions, subtractions, and multiplications, and S_{f_i} with vector multiplications and cyclic/negacyclic shifts.

Dimension requirement of vectorization-friendliness. From the definition, we know that f is vectorization-friendly only if its domain has dimension a multiple of v.

Additional properties of vectorization-friendliness. Obviously, if an algebra homomorphism is vectorization-friendly, its inverse (if exists) and module-theoretic dual are also vectorization-friendly.

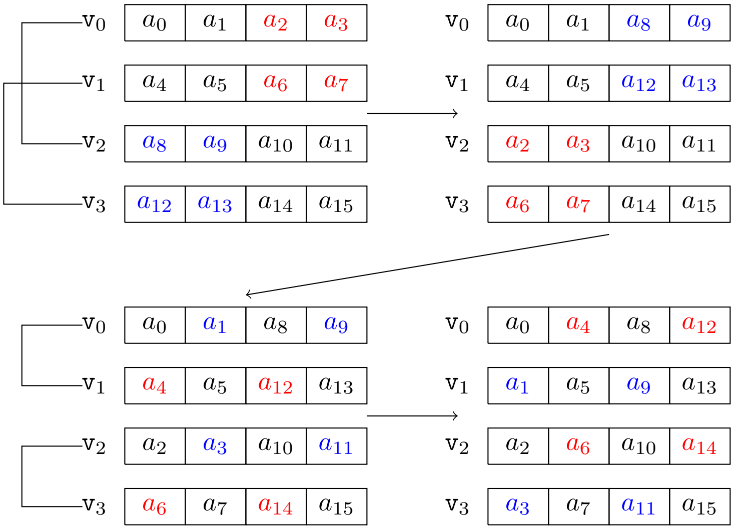
7.2 Permutation-Friendliness

Conceptually, permutation-friendliness stands for vectorization-friendliness up to a special type of permutation – interleaving.

7.2.1 Transpositions and Interleaving

We give a conceptual review of transposing a matrix with vector instructions. The idea was introduced by [Flo72] under the context of permuting with pages and more recently by [War12, Section 7.3] for permuting bit matrices. It was also used in [NG21, BBCT22, BHK+21, CCHY24, Hwa24c, HLY24] for vectorized polynomial multiplications.

Figure 7.1: Top-down transposition of the 4×4 matrix with rows $(a_0, \ldots, a_3)$, $(a_4, \ldots, a_7)$, $(a_8, \ldots, a_{11})$, and $(a_{12}, \ldots, a_{15})$.



$\mathtt{v}_0$	a_0	a_1	a_2	a_3
$\mathtt{v}_1$	a_4	a_5	a_6	a_7
$\mathtt{v}_2$	a_8	a_9	a_{10}	a_{11}
$\mathtt{v}_3$	a_{12}	a_{13}	a_{14}	a_{15}

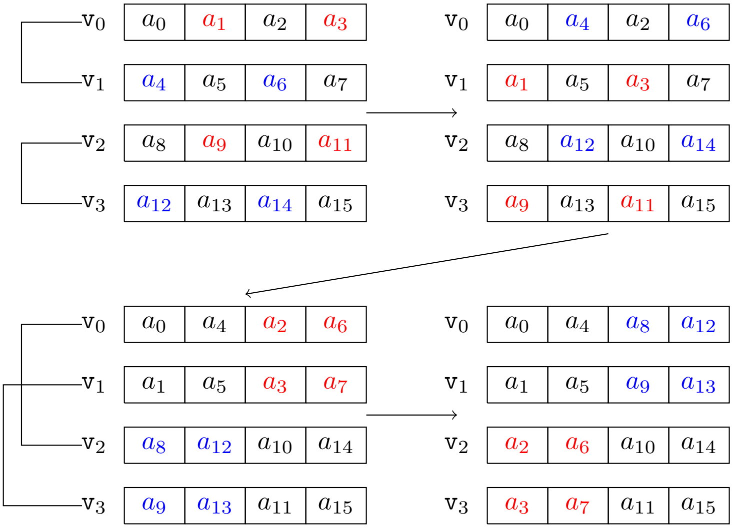
$\mathtt{v}_0$	a_0	a_1	a_8	a_9
$\mathtt{v}_1$	a_4	a_5	a_{12}	a_{13}
$\mathtt{v}_2$	a_2	a_3	a_{10}	a_{11}
$\mathtt{v}_3$	a_6	a_7	a_{14}	a_{15}

$\mathtt{v}_0$	a_0	a_1	a_8	a_9
$\mathtt{v}_1$	a_4	a_5	a_{12}	a_{13}
$\mathtt{v}_2$	a_2	a_3	a_{10}	a_{11}
$\mathtt{v}_3$	a_6	a_7	a_{14}	a_{15}

$\mathtt{v}_0$	a_0	a_4	a_8	a_{12}
$\mathtt{v}_1$	a_1	a_5	a_9	a_{13}
$\mathtt{v}_2$	a_2	a_6	a_{10}	a_{14}
$\mathtt{v}_3$	a_3	a_7	a_{11}	a_{15}

For simplicity, we illustrate how to transpose a 4×4 matrix. Suppose the matrix is represented in the row-major fashion by four vectors $\mathtt{v}_0, \ldots, \mathtt{v}_3$ with each holding four elements. There are two steps: (i) transpose as if we are transposing a 2×2 matrix with each entries 2×2 matrix, and (ii) transpose each of the 2×2 matrices. We implement step (i) by permuting the pairs $(\mathtt{v}_0, \mathtt{v}_2)$ and $(\mathtt{v}_1, \mathtt{v}_3)$, and for step (ii), we permute the pairs $(\mathtt{v}_0, \mathtt{v}_1)$ and $(\mathtt{v}_2, \mathtt{v}_3)$. See Figure 7.1 for an illustration. Obviously, the idea generalizes to transposing arbitrary $2^k \times 2^k$ matrices – we transpose as if we are transposing a 2×2 matrix with entries $2^{k-1} \times 2^{k-1}$ matrices, and transpose the $2^{k-1} \times 2^{k-1}$ matrices recursively. Since we start from the root level of the recursion tree, we call the approach top-down transposition. Notice that we can swap the order of (i) and (ii). We call the resulting approach bottom-up transposition (cf. Figure 7.2).

Figure 7.2: Bottom-up transposition of the 4×4 matrix with rows $(a_0, \ldots, a_3)$, $(a_4, \ldots, a_7)$, $(a_8, \ldots, a_{11})$, and $(a_{12}, \ldots, a_{15})$.



The set `Interleave` [Hwa24c]. Again, let v' be a multiple of v. We define the **transposition matrix** $T_{v'^2}$ as the $v'^2 \times v'^2$ matrix permuting the elements as if transposing a $v' \times v'$ matrix. We call a matrix M **interleaving matrix with step** v' if it takes the form

$$M = (\pi' \otimes I_{v'}) \, (I_m \otimes T_{v'^2}) \, (\pi \otimes I_{v'})$$

for a positive integer m and permutation matrices π, π' permuting mv' elements. The set `Interleave` consists of interleaving matrices of all possible steps. Obviously, we can implement an interleaving matrix as a transposition matrix with on-the-fly permutations.

7.2.2 Permutation-Friendliness, Formally

Let g be an algebra homomorphism and M_g its matrix form. We call g **permutation-friendly** if

$$M_g = \prod_i S_{g_i} M_{g_i}$$

for $S_{g_i} \in$ `Interleave` and vectorization-friendly M_{g_i}s. Once we find such a decomposition of a permutation-friendly g, we implement the vectorization-friendly parts as described in previous section and the interleaving matrices with permutation instructions.

Dimension requirement of permutation-friendliness. By definition, permutation-friendliness necessitates a stronger condition on the dimension than vectorization-friendliness due to the existence of interleaving matrices. Interleaving matrices necessitates that a permutation-friendly homomorphism must have dimension a multiple of v^2, and the condition is strictly stronger than vectorization-friendliness whenever $v > 1$.

7.3 Vectorizing Toeplitz Matrix-Vector Product

For the small-dimensional Toeplitz matrix-vector products, [CCHY24] showed that one can implement Toeplitz matrix-vector products efficiently with vector-by-scalar multiplication instructions. For simplicity, we demonstrate with the case $m = n = 4$ and $R = \mathbb{Z}_{2^{32}}$:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_0 & a'_1 & a'_2 & a'_3 \\ a_1 & a_0 & a'_1 & a'_2 \\ a_2 & a_1 & a_0 & a'_1 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} .$$

For deploying vector-by-scalar multiplications, the key is to identify the reuses of the scalar operands. Obviously, each of $b_0, \dots, b_3$ is involved in four multiplications in R, and we map each columns of the matrix to a vector and apply vector-by-scalar multiplications. There are two ways for constructing the column vectors of $\mathbf{Toeplitz}(a_3, \cdots, a_0, a'_1, \cdots, a'_3)$ from an array storing $a'_3, \dots, a'_1, a_0, \dots, a_3$: either loading from the addresses pointing to $a_0, a'_1, \dots, a_3$, or loading the first column and first row and combining them with special instructions. After constructing the matrix column-wise, we now identify the resulting column vector as the sum of columns scaled by the corresponding elements in $\boldsymbol{b}$. In other words,

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = b_0 \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} + b_1 \begin{pmatrix} a'_1 \\ a_0 \\ a_1 \\ a_2 \end{pmatrix} + b_2 \begin{pmatrix} a'_2 \\ a'_1 \\ a_0 \\ a_1 \end{pmatrix} + b_3 \begin{pmatrix} a'_3 \\ a'_2 \\ a'_1 \\ a_0 \end{pmatrix} .$$

Algorithm 7.1 is an illustration. The above observation obviously scales to a Toeplitz matrix-vector product with dimension a multiple of v.

Algorithm 7.1 Applying a 4 × 4 Toeplitz matrix with vector-by-scalar multiplication instructions [CCHY24].

Inputs: **Toeplitz**$(a_3, a_2, a_1, a_0, a'_1, a'_2, a'_3)$, (b_0, b_1, b_2, b_3).
Outputs: **Toeplitz**$(a_3, a_2, a_1, a_0, a'_1, a'_2, a'_3)(b_0, b_1, b_2, b_3)$.

```
t0 = $a_3||a_2||a_1||a_0$
t1 = $a_2||a_1||a_0||a'_1$
t2 = $a_1||a_0||a'_1||a'_2$
t3 = $a_0||a'_1||a'_2||a'_3$
c = mul(t0, $b_0$)
c = mla(c, t1, $b_1$)
c = mla(c, t2, $b_2$)
c = mla(c, t3, $b_3$)
```

7.4 Choosing Homomorphisms for Vectorization

We go through the overall vectorization framework for polynomial multiplications in $R[x]/\langle x^n - \alpha x - \beta \rangle$.

7.4.1 With Vector-by-Scalar Multiplication Instructions

The first thing is to figure out if there are vector-by-scalar multiplication instructions implementing the small-power-of-two-dimensional Toeplitz matrix-vector products over the ring R. Assuming $R = \mathbb{Z}_q$, if $q = 2^k$, we seek for vector-by-scalar low multiplication instructions. On the other hand, if q contains an odd factor, we seek for vector-by-scalar long multiplication instructions so we can compute the sums of long products.

The next step is to choose a vectorization-friendly monomorphism f computing $\boldsymbol{ab} = f^{-1}(f(\boldsymbol{a})f(\boldsymbol{b}))$. If f results in small-power-of-two-dimensional Toeplitz matrix-vector products, we implement the Toeplitz matrix-vector products with vector-by-scalar multiplication instructions. And if f results in small-power-of-two-dimensional polynomial multiplications that are not Toeplitz matrix-vector products, we identify the bilinear map g implementing a Toeplitz matrix-vector product as $f^* \circ g\left(f(-), f^{-1*}\right)$. Since polynomial multiplications in $R[x]/\langle x^n - \alpha x - \beta \rangle$ can be computed with Toeplitz matrix-vector products of matrix dimension $n \times n$ (cf. Section 6.5), we eventually have small-dimensional Toeplitz matrix-vector products.

7.4.2 With Vector-by-Vector Multiplication Instructions

If there are no suitable vector-by-scalar multiplication instructions implementing small-power-of-two-dimensional Toeplitz matrix-vector products over R, we choose a vectorization-friendly f and a permutation-friendly g computing $\boldsymbol{ab} = (g \circ f)^{-1} ((g \circ f) (\boldsymbol{a}) (g \circ f) (\boldsymbol{b}))$, and implement everything with vector-by-vector multiplication instructions.

Part II

General Guide for Optimizations

Chapter 8

Platforms

8.1 Instruction Set Architectures and Extensions

An instruction set architecture (ISA) specifies how software control the processing units. In an ISA, we have registers holding the data, several instructions transferring data between memory and registers, and several instructions processing the data. To simplify the descriptions of ISAs, this thesis abbreviates the instructions as follows. Suppose we have instructions `add`, `sub`, `sadd`, `ssub`, `uadd`, `usub`. `{, u, s}{add, sub}` stands for "`add`, `sub`, `sadd`, `ssub`, `uadd`, or `usub`" or "`add`, `sub`, `sadd`, `ssub`, `uadd`, and `usub`," depending on the context.

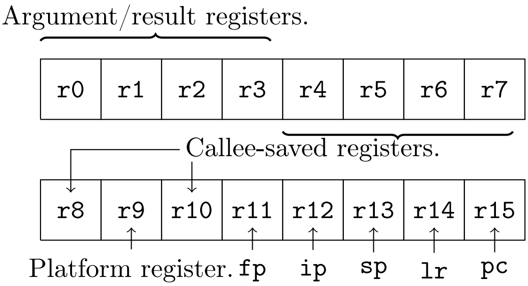
8.1.1 Armv7-M and Armv7E-M

Armv7-M is a 32-bit ISA. It comes with optional Digital Signal Processing (DSP) extension, single-precision floating-point extension, and double-precision floating-point extension. If DSP extension is implemented, the ISA is usually termed Armv7E-M. There are two optional floating-point extensions: FPv4-SP for single-precision and FPv5 for double precision.

8.1.1.1 General-Purpose Registers

Registers and procedure call standard [ARM21d, Section 6.1.1]. There are 16 general-purpose registers: `r0`, . . . , `r15` [ARM21b, Section A2.3.1]. The first four registers `r0`, . . . , `r3` are argument and result registers. Register `r9` is the platform register, register `r11` is the frame pointer `fp` if presented or as a callee-saved register, register `r12` is the intra-procedure-call register `ip`, register `r13` is the stack pointer `sp`, register `r14` is the link register `lr`, and register `r15` is the program counter `sp`. The remaining registers `r4`, . . . , `r8`, `r10` are callee-saved registers. In general, `r11` is regarded as a callee-saved register if we compile the programs while omitting the frame pointer, `r12` is regarded as a temporary register if we manage the intra-procedure calls ourselves, and `r14` is regarded as a callee-saved register. Frequently, `r9` is also regarded as a callee-saved register when system register is not needed. See Figure 8.1 for an illustration.

Figure 8.1: General-purpose registers in Armv7-M architecture [Hwa22, ARM21b]. Each rectangle represents a 32-bit register.



8.1.1.2 Instructions

Single-register load/store. For single-register load, `ldrb` loads an 8-bit value and zero-extends it, `ldrsb` loads an 8-bit value and sign-extends it, `ldrh` loads a 16-bit value and zero-extends it, `ldrsh` loads a 16-bit value and sign-extends it, and `ldr` loads a 32-bit value. For single-register store, `strb` stores an 8-bit value, `strh` stores a 16-bit value, and `str` stores a 32-bit value.

Two-register load/store. For two-register load and store, `ldrd` loads two consecutive 32-bit values and `strd` stores two 32-bit values to a 64-bit memory space.

Multi-register load/store. For multi-register load and store, `ldm` loads several consecutive 32-bit values and `strd` stores several 32-bit values to consecutive memory space. Memory operations permit pre-indexed and post-indexed updates to the base register. See Table 8.1 for a summary and [ARM21b, Tables A4-17 and A4-18] for full lists of memory operations.

Table 8.1: Memory operations in Armv7-M [ARM21b, Tables A4-17 and A4-18].

	Signed			Unsigned			Reg. set
	8-bit	16-bit	32-bit	8-bit	16-bit	32-bit	
Load	`ldrb`	`ldrh`	`ldr`	`ldrsb`	`ldrsh`	`ldr`	`ldm`
Store	`strb`	`strh`	`str`	`strb`	`strh`	`str`	`stm`

Byte-level permutations. For byte-level permutations, `rev` reverses the byte order of the 32-bit value, `rev16` reverses the byte order of each 16-bit values in the 32-bit register, and `revsh` reverses the byte order of the lowest 16 bits and sign-extends it to a 32-bit value [ARM21b, Table A4-15].

Additions and subtractions. For additions and subtractions, `add` adds up two 32-bit values, `adc` adds up two 32-bit values with carry, `sub` subtracts two 32-bit values, `sbc` subtracts two 32-bit values with carry, `rsb` subtracts two 32-bit values with operands swapped, and `neg` negates a 32-bit value and is implemented with `rsb` [ARM21b, Table A4-2].

Comparisons. For comparisons, `cmp` subtracts an immediate value or a register from a register, updates the conditional flags, and discards the result, and `cmn` adds up a register value with an immediate value or a register instead [ARM21b, Table A4-2].

Bitwise operations. For bitwise operations, we have bitwise copy `mov`, bitwise not `mvn`, bitwise and `and`, bitwise clear `bic`, bitwise or `orr`, bitwise or-not `orn`, and bitwise exclusive-or `eor` [ARM21b, Tables A4-2 and A4-15]. `bic` applies `and` to the first 32-bit value and the bitwise not of the second 32-bit value. This effectively clears the corresponding bits of the first 32-bit value. `orn` applies `orr` to the first 32-bit value and the bitwise not of the second 32-bit value.

Bit-field operations. For bit-field operations, `bfc` clears the specified consecutive bits and `bfi` inserts the specified number of the lowest bits to the specified position [ARM21b, Table A4-15]. `uxtb` zero-pads the lowest 8 bits to a 32-bit value, `uxth` zero-pads the lowest 16 bits to a 32-bit value, and `ubfx` extracts the specified consecutive bits and zero-pads extracted bits to a 32-bit value. `sxt{b, h}`, and `sbfx` are the signed counterparts sign-extending to 32-bit values [ARM21b, Tables A4-12 and A4-15]. `rbit` reverses the 32-bit value [ARM21b, Table A4-15].

Shift operations. For shift operations, `lsl` left-shift the 32-bit value with zeros, `lsr` right-shift the 32-bit value with zeros, `asr` right-shift the 32-bit value with the 31st bit, and `ror` rotates the 32-bit [ARM21b, Table A4-3].

Multiplications. For multiplications, `mul` multiplies two 32-bit values and return the lowest 32 bits, `mla` applies `mul` and accumulates the result to a 32-bit value, and `mls` applies `mul` and subtracts the result from a 32-bit value [ARM21b, Table A4-4]. `umull` multiplies two unsigned 32-bit values and return the resulting 64-bit value to two 32-bit registers, `umlal` applies `umull` and accumulates the result to the 64-bit value represented by two 32-bit registers [ARM21b, Table A4-7]. `s{mul, mla}l` are the signed counterparts of `u{mul, mla}l` [ARM21b, Table A4-5].

8.1.1.3 Digital Signal Processing Extension

Table 8.2: 8-bit parallel additions and subtractions in the DSP extension of Armv7E-M [ARM21b, Table A4-14].

	Signed		Unsigned	
	Addition	Subtraction	Addition	Subtraction
Standard	sadd8	ssub8	uadd8	usub8
Halved	shadd8	shsub8	uhadd8	uhsub8
Saturated	qadd8	qsub8	uqadd8	uqsub8

8-bit parallel additions and subtractions. For 8-bit parallel additions and subtractions, each 32-bit register is treated as four 8-bit values. `uadd8` component-wisely adds the 8-bit values and places the resulting 8-bit values to the destination register, `uqadd8` saturates the results to $[0, 2^8)$, and `uhadd8` halves the results. `u{, q, h}sub8` are the subtractive version, and `{s,` `q, sh}{add8, sub8}` are the signed counterparts where the saturated ones saturate the results to $[-2^7, 2^7)$ [ARM21b, Table A4-14]. See Table 8.2 for a summary.

16-bit parallel additions and subtractions. For 16-bit parallel additions and subtractions, each 32-bit register is treated as two 16-bit values. We have the component-wise additions and subtractions `u{, q, h}{add16, sub16}` and `{s, q, sh}{add16, sub16}` as in the 8-bit case. See Table 8.3 for a summary. We also have several "`x`" ones with a pair of addition and subtraction: `uasx` exchanges the 16-bit values in the second source register, adds up the the first 16-bit values, subtracts the second 16-bit values, and places the resulting 16-bit values to the destination register. Similarly, `uqasx` saturates the results and `uhasx` halves the results. `u{, q, h}sax` subtract the first 16-bit values and add the second 16-bit values instead. `{s, q, sh}{as, sa}x` are the signed counterparts [ARM21b, Table A4-14]. See Table 8.4 for a summary.

Table 8.3: 16-bit parallel additions and subtractions in the DSP extension of Armv7E-M [ARM21b, Table A4-14].

	Signed		Unsigned	
	Addition	Subtraction	Addition	Subtraction
Standard	`sadd16`	`ssub16`	`uadd16`	`usub16`
Halved	`shadd16`	`shsub16`	`uhadd16`	`uhsub16`
Saturated	`qadd16`	`qsub16`	`uqadd16`	`uqsub16`

Table 8.4: 16-bit crossed additions and subtractions in the DSP extension of Armv7E-M [ARM21b, Table A4-14].

	Signed		Unsigned	
	Add-sub	Sub-add	Add-sub	Sub-add
Standard	`sasx`	`ssax`	`uasx`	`usax`
Halved	`shasx`	`shsax`	`uhasx`	`uhsax`
Saturated	`qasx`	`qsax`	`uqasx`	`uqsax`

Signed 16-bit multiplications. For signed 16-bit multiplications, `smul{b, t}{b, t}` multiplies two 16-bit values and returns the 32-bit result. The first `{b, t}` specifies a 16-bit value in the first register and the second `{b, t}` specifies a 16-bit value in the second register. The symbol `b` specifies the lowest 16-bit value as an operand and the symbol `t` specifies the highest 16-bit value as an operand. `smla{b, t}{b, t}` multiplies two 16-bit values and accumulates the result to the destination register. `smlal{b, t}{b, t}` multiplies two 16-bit values, sign-extends the result to 64-bit, and accumulates the result to two destination registers [ARM21b, A4-6]. See Table 8.5 for a summary.

Table 8.5: Signed 16-bit multiplications in the DSP extension of Armv7E-M [ARM21b, Table A4-6].

	Bit-size		
	Operands	Result	Accumulator
`smul{b, t}{b, t}`	$(16, 16)$	32	-
`smla{b, t}{b, t}`	$(16, 16)$	32	32
`smlal{b, t}{b, t}`	$(16, 16)$	32	64

Signed dual 16-bit multiplications. For signed dual 16-bit multiplications, each instructions performs two 16-bit multiplications and adds or subtracts the 32-bit products. The symbol `x` is optional and exchanges the 16-bit values of the second source register. `smuad{, x}` performs two 16-bit multiplications, adds the products, and places the 32-bit result to the destination register. `smusd{, x}` subtracts the second product from the first product. `smlad{, x}` adds the products and accumulates the result to the destination register. `smlsd{, x}` subtracts the result from the destination register. `smlald{, x}` performs two 16-bit multiplications, adds the products, sign-extends the result to 64-bit, and accumulates the 64-bit result to two destination registers. `smlsld{, x}` subtracts the 64-bit result from the two destination registers [ARM21b, Table A4-6]. See Table 8.6 for a summary.

Table 8.6: Signed dual 16-bit multiplications in the DSP extension of Armv7E-M [ARM21b, Table A4-6].

	Bit-size		
	Operands	Result	Accumulator
`smu{a, s}d{, x}`	$(16, 16)$	32	-
`sml{a, s}d{, x}`	$(16, 16)$	32	32
`sml{a, s}ld{, x}`	$(16, 16)$	64	64

Signed wide multiplications. For signed wide multiplications, `smulw{b, t}` multiplies a 32-bit value by a 16-bit value specified by the symbol `{b, t}`, extracts the most significant 32-bit value of the 48-bit product. and places the 32-bit result to the destination register. `smlaw{b, t}` accumulates the 32-bit result to the destination register [ARM21b, Table A4-6]. See Table 8.7 for a summary.

Table 8.7: Signed wide 16-bit multiplications in the DSP extension of Armv7E-M [ARM21b, Table A4-6].

	Bit-size		
	Operands	Result	Accumulator
`smulw{b, t}`	$(32, 16)$	32	-
`smlaw{b, t}`	$(32, 16)$	32	32

Signed most significant-word multiplications. For signed most significant-word multiplications, `smmul{, r}` multiplies two 32-bit values, extracts the highest 32 bits of the 64-bit product, and places the 32-bit result to the destination register. The symbol `r` is optional and rounds the 64-bit product while extracting the highest 32 bits. `smmla{, r}` left-shifts the 32-bit accumulator by 32 bits producing a 64-bit accumulator, accumulates the 64-bit product to the 64-bit accumulator, and extracts the highest 32 bits with optional rounding. `smmls{, r}` subtracts the 64-bit product from the 64-bit accumulator instead [ARM21b, Table A4-6]. See Table 8.8 for a summary.

Table 8.8: Signed most significant-word multiplications in the DSP extension of Armv7E-M [ARM21b, Table A4-6].

	Bit-size		
	Operands	Result	Accumulator
`smmul{, r}`	$(32, 32)$	32	-
`smmla{, r}`	$(32, 32)$	32	32
`smmls{, r}`	$(32, 32)$	32	32

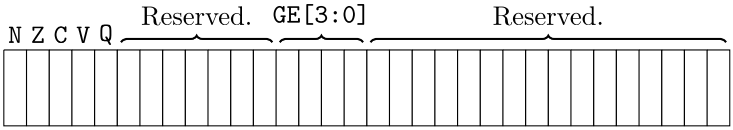
Unsigned multiplication. For unsigned long multiplication, `umaal` computes $ab + c + d$ for 32-bit values $a, b, c, d \in \left[0, 2^{32}\right) \cap \mathbb{Z}$ [ARM21b, Table A4-8]. Since $ab + c + d \leq \left(2^{32} - 1\right)^2 + 2\left(2^{32} - 1\right) = 2^{64} - 1$, `umaal` never carries.

16-bit Pack instructions. For 16-bit pack instructions, we have `pkhbt` and `pkhtb` packing a 16-bit value of the first source register and a 16-bit value of the second source register. `pkhbt` packs the lowest 16 bits of the first and the highest 16 bits of the second, and `pkhtb` packs the highest 16 bits of the first and the lowest 16 bits of the second [ARM21b, Table A4-13]. The 16-bit value of the first is mapped to the lowest 16 bits of the destination register, and the 16-bit value of the second is mapped to the highest 16 bits of the destination register.

8.1.1.4 Application Program Status Register

There is a 32-bit application program status register (APSR) holding the program status [ARM21b, Section A2.3.2]. Starting from the most significant bit, we have the 1-bit negative condition flag `N`, 1-bit zero condition flag `Z`, 1-bit carry condition flag `C`, 1-bit overflow condition flag `V`, 1-bit saturation flag `Q`, 7 reserved bits, 4-bit SIMD greater than or equal to flag `GE[3:0]`, and 16 reserved bits. The 4-bit `GE[3:0]` is implemented only in the DSP extension, and is reserved on platforms without the DSP extension. See Figure 8.2 for an illustration.

Figure 8.2: Application program status register in Armv7-M adapted from [ARM21b, Section A2.3.2]. Each rectangle represents a bit.



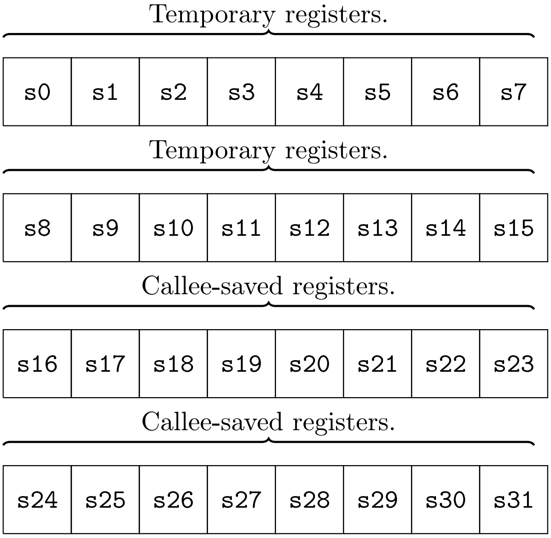
Conditional flags and conditional executions. The conditional flags together implement several mnemonics: the equal mnemonic `EQ`, the not-equal mnemonic `NE`, the carry set mnemonic `CS`, the carry clear mnemonic `CC`, the minus mnemonic `MI`, the positive-or-zero mnemonic `PL`, the overflow mnemonic `VS`, the no-overflow mnemonic `VC`, the unsigned higher mnemonic `hi`, the unsigned lower-or-same mnemonic `ls`, the signed greater-than-or-equal mnemonic `ge`, the signed greater-than mnemonic `gt`, the signed less-than-or-equal mnemonic `le`, and the signed less-than mnemonic `lt` [ARM21b, Table 7-1].

8-bit selection. The 8-bit selection `sel` is part of the DSP extension and it selects four 8-bit values from two source 32-bit registers according to the 4-bit `GE[3:0]` flag [ARM21b, Section A4-16]. For each of the bits in `GE[3:0]`, if it is 1, then the corresponding 8-bit value of the first source register is selected; otherwise the corresponding 8-bit value of the second source register is selected.

8.1.1.5 FPv4-SP Extension

Registers and procedure call standard [ARM21d, Section 6.1.2]. Armv7-M comes with optional floating-point extensions: FPv4-SP for single-precision extension and FPv5 for double-precision extension. Cortex-M4 optionally supports FPv4-SP. There are 32 single-precision floating-point registers: `s0`, . . . , `s31`. Registers `s0`, . . . , `s15` are temporary registers, and registers `s16`, . . . , `s31` are callee-saved registers. See Figure 8.3 for an illustration. We do not rely on floating-point arithmetic. However, floating-point registers are used as low-latency cache [ACC$^+$20] due to the fast transfer between floating-point and general-purpose registers on Cortex-M4 [ARM21b, Section A4.12].

Figure 8.3: Floating-point registers in FPv4-SP extension of Armv7-M. Each rectangle represents a single-precision floating-point register.



Floating-point transferring operations. For floating-point transferring operations, `vmov` transfers between floating-point and general-purpose/floating-point registers [ARM21b, Section A4-21]. For single-precision floating-point values, we can transfer the content of a single-precision floating-point register to another single-precision floating-point register or a general-purpose register and vice versa. For double-precision floating-point values, we can transfer the contents of two consecutive single-precision floating-point register to two general-purpose registers and vice versa. See Table 8.9 for a summary.

Table 8.9: Floating-point transferring operations in the FPv4-SP of Armv7-M [ARM21b, Table 7-1].

Destination	Source
`vn`	`vm`
`vn`	`rm`
`rn`	`vm`
`vn`, `v(n + 1)`	`rm0`, `rm1`
`rm0`, `rm1`	`vn`, `v(n + 1)`

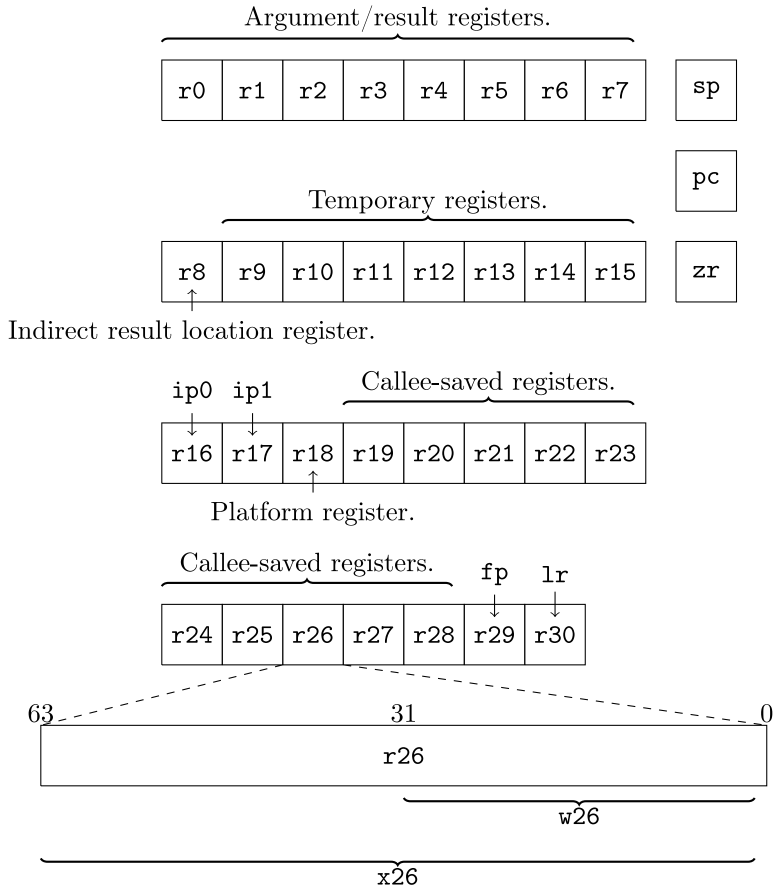
8.1.2 Armv8-A Neon

Armv8-A is an ISA targeting application uses. The ISA is implemented in a variety of smartphones and personal computers. Section 8.1.2.1 reviews the general-purpose registers of the ISA. It also comes with the SIMD technology Neon. Sections 8.1.2.2 and 8.1.2.3 reviews the SIMD technology Neon and corresponding instructions.

8.1.2.1 General-Purpose Registers

Registers and procedure call standard [ARM21c, Section 6.1.1]. In Armv8-A, there are 31 64-bit general-purpose registers: `r0`, . . . , `r30`. Each registers can be accessed as a 64-bit register `xn` or a 32-bit register `wn` [ARM21a, Section B1.2.1]. The 32-bit name `wn` is also used as an 8-bit or a 16-bit operand in some instructions. Registers `r0`, . . . , `r7` hold the arguments and the results of a function. Register `r8` is an indirect result location register passing the address of an indirect result. Registers `r9`, . . . , `r15` are temporary registers. Registers `r16` and `r17` are intra-procedural-call registers when labeled as `ip0` and `ip1`, and may be used as temporary registers. Register `r18` is a platform register and shall not be used in platform-independent programs. Registers `r19`, . . . , `r28` are callee-saved registers. Register `r29` is the frame pointer labeled as `fp` and `r30` is the link register labeled as `lr`. There are also a stack pointer `sp` accessed as `wsp` and `xsp`, a program counter `pc`, and a zero register `zr` accessed as `wzr` and `xzr`. The zero register `zr` is not necessary implemented as a physical register [ARM21a, Section B1.2.1]. See Figure 8.4 for an illustration.

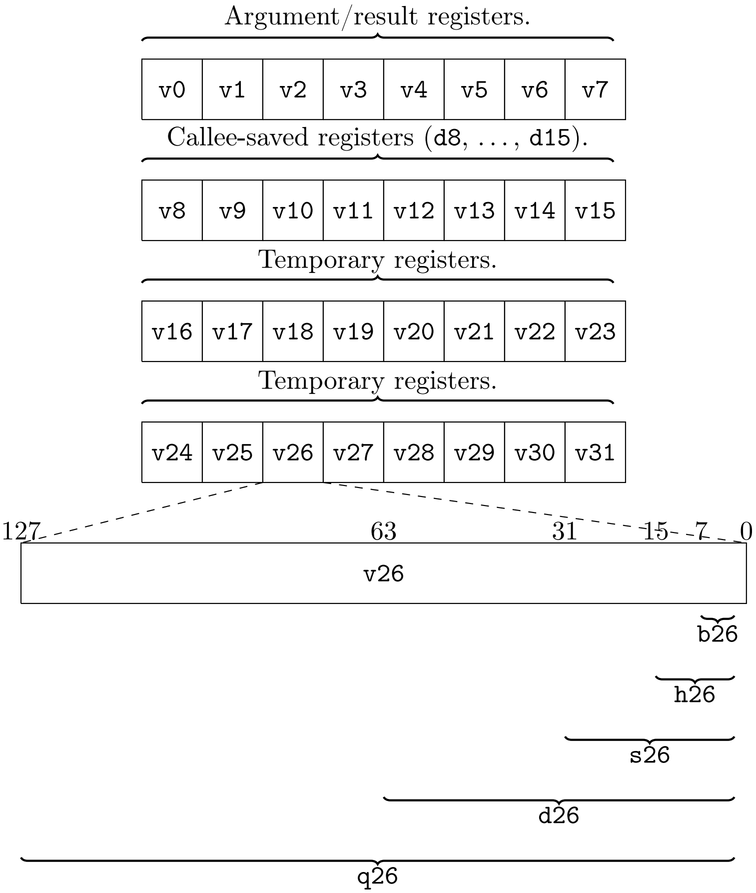
Figure 8.4: General-purpose registers in Armv8-A [ARM21a]. Each rectangle represents a 64-bit register.



8.1.2.2 SIMD Registers

Registers and procedure call standard [ARM21c, Section 6.1.2]. In the Armv8-A SIMD technology Neon, there are 32 128-bit SIMD registers: `v0`, . . . , `v31`. Each SIMD register can be accessed as an 8-bit register `bn`, a 16-bit register `hn`, a 32-bit register `sn`, a 64-bit register `dn`, a 128-bit register `qn`, or a 128-bit SIMD register `vn`. The first eight registers `v0`, ..., `v7` are argument and result registers, the follow-up eight registers `v8`, ..., `v15` are callee-saved registers, and the rests are temporary registers. For the callee-saved registers `v8`, ..., `v15`, we only need to preserve the bottom 64-bit `d8`, ..., `d15`. See Figure 8.5 for an illustration.

Figure 8.5: SIMD registers in Armv8-A Neon [ARM21a]. Each rectangle represents a 128-bit register.



8.1.2.3 Neon Instructions

We abbreviate the operands as follows. `w` and `x` refer to the 32-bit and 64-bit general-purpose register; `B`, `H`, `S`, and `D` refer to the single-register name (`bn`, `hn`, `sn`, `dn`) of a SIMD register; `8B`, `16B`, `4H`, `8H`, `2S`, `4S`, and `2D` refer to the packed format of a SIMD register; `B[]`, `H[]`, `S[]`, and `D[]` refer to a certain lane of the packed format of a SIMD register. "Vector" stands for `8B/16B/4H/8H/2S/4S/2D`, `immX` stands for an `X`-bit immediate, and 0 stands for the value 0.

Loads and stores. For loads and stores, `ldr` loads an 8-bit value, a 16-bit value, a 32-bit value, a 64-bit value, or a 128-bit value, and `str` stores an 8-bit value, a 16-bit value, a 32-bit value, a 64-bit value, or a 128-bit value. See [ARM21a, Section C3.2.10] for a full list of loads and stores.

Transferring operations. For transferring elements between registers, we have operations `dup` and `mov` duplicating an element from a general-purpose register or a lane of a SIMD register to a general-purpose register, a lane of a SIMD register, or a whole SIMD register, and a SIMD register to a SIMD register [ARM21a, Tables C3-79 and C3-80]. See Table 8.10 for a summary of operands of transferring operations.

Table 8.10: Summary of operands of transferring operations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`dup`	`B/H/S/D`	`w/x`
`dup`	`B/H/S/D`	`B[]/H[]/S[]/D[]`
`dup`	Vector	`B[]/H[]/S[]/D[]`
`mov`	`B[]/H[]/S[]/D[]`	`w/x`
`mov`	`w/x`	`S[]/D[]`
`mov`	`B/H/S/D`	`B[]/H[]/S[]/D[]`
`mov`	`B[]/H[]/S[]/D[]`	`B[]/H[]/S[]/D[]`
`mov`	`8B/16B`	`8B/16B`

Permutations. For permuting the elements, we have operations `ext`, `trn{1, 2}`, `uzp{1, 2}`, `zip{1, 2}` [ARM21a, Table C3-85]. `ext` concatenates two SIMD registers, extracts 16 consecutive bytes, and places the results to the destination SIMD register. `trn{1, 2}` extracts 8 non-consecutive bytes from two source registers and interleaves them. `trn1` interleaves the odd-indexed ones and `trn2` interleaves the even-indexed ones. `uzp{1, 2}` extracts 8 non-consecutive bytes from two source registers and concatenates them. `uzp1` extracts the even-indexed ones from the first and odd-indexed ones from the second, and `uzp2` extracts the odd-indexed ones from the first and even-indexed ones from the second. `zip{1, 2}` extracts 8 consecutive bytes from two source registers and interleaves them. `zip1` extracts the lowest 8 bytes of each register and `zip2` extracts the highest 8 bytes of each register. See Table 8.11 for comparisons of `trn{1, 2}`, `uzp{1, 2}`, and `zip{1, 2}`, and Table 8.12 for a summary of operands of permutations. Notice that interleaving implies elements from the same register are written to the destination register non-consecutively.

Table 8.11: Comparisons of permutations in Armv8-A Neon [ARM21a].

Instruction	Input elements	Output elements
`trn{1, 2}`	Non-consecutive.	Non-consecutive.
`uzp{1, 2}`	Non-consecutive.	Consecutive.
`zip{1, 2}`	Consecutive.	Non-consecutive.

Table 8.12: Summary of operands of permutations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`ext`	`8B/16B`	`8B/16B`, `imm4`
`{trn, uzp, zip}{1, 2}`	Vector	Vector

Additions and subtractions. For additions and subtractions, we have the standard ones `add`, `sub`, and `neg` [ARM21a, Tables C3-80 and C3-83] and several variants. `sqneg` saturates the results [ARM21a, Table C3-83]. `{u, s}{h, q}{add, sub}` halves or saturates the unsigned or signed results [ARM21a, Table C3-80]. `{u, s}rhadd` halves with rounding the unsigned or signed results [ARM21a, Table C3-80]. `{, r}{add, sub}hn{, 2}` computes the double-size results, extracts the highest halves of each elements with optional rounding, and places the results at the lowest or the highest half [ARM21a, Table C3-82]. `{u, s}{add, sub}l{, 2}` computes the double-size results of the lowest or the highest halves, `{u, s}{add, sub}w{, 2}` adds or subtracts the elements of the lowest or the highest half of the second source register from the first source register [ARM21a, Table C3-82]. See Table 8.13 for a summary of operands of additions and subtractions.

Table 8.13: Summary of operands of additions and subtractions in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`add/sub`	`B/H/S/D`	`B/H/S/D`
`add/sub`	Vector	Vector
`neg/sqneg`	`B/H/S/D`	`B/H/S/D`
`neg/sqneg`	Vector	Vector
`{u, s}hadd`	`8B/16B/4H/8H/2S/4S`	`8B/16B/4H/8H/2S/4S`
`{u, s}hsub`	`8B/16B/4H/8H/2S/4S`	`8B/16B/4H/8H/2S/4S`
`{u, s}qadd`	`B/H/S/D`	`B/H/S/D`
`{u, s}qadd`	Vector	Vector
`{u, s}qsub`	`B/H/S/D`	`B/H/S/D`
`{u, s}qsub`	Vector	Vector
`{u, s}rhadd`	`8B/16B/4H/8H/2S/4S`	`8B/16B/4H/8H/2S/4S`
`{, r}haddhn`	`8B/4H/2S`	`8H/4S/2D`
`{, r}haddhn2`	`16B/8H/4S`	`8H/4S/2D`
`{, r}hsubhn`	`8B/4H/2S`	`8H/4S/2D`
`{, r}hsubhn2`	`16B/8H/4S`	`8H/4S/2D`
`{u, s}addl`	`8H/4S/2D`	`8B/4H/2S`
`{u, s}addl2`	`8H/4S/2D`	`16B/8H/4S`
`{u, s}subl`	`8H/4S/2D`	`8B/4H/2S`
`{u, s}subl2`	`8H/4S/2D`	`16B/8H/4S`
`{u, s}addw`	`8H/4S/2D`	`8H/4S/2D`, `8B/4H/2S`
`{u, s}addw2`	`8H/4S/2D`	`8H/4S/2D`, `16B/8H/4S`
`{u, s}subw`	`8H/4S/2D`	`8H/4S/2D`, `8B/4H/2S`
`{u, s}subw2`	`8H/4S/2D`	`8H/4S/2D`, `16B/8H/4S`

Widening and narrowing operations. For narrowing and widening operations, we have the narrowing operations `xtn{, 2}`, `{u, s}qxtn{, 2}` and the widening operations `{u, s}xtl{, 2}` [ARM21a, Table C3-83]. `xtn` extracts the elements of the source register, halves the sizes, places the results to the lowest half of the destination register, and clears the highest half of the destination register. `xtn2` places the results to the highest half of the destination register where the lowest half is untouched. `{u, s}qxtn` saturates the unsigned or signed results, places the results to the lowest half of the destination register, and clears the highest half of the destination register. `{u, s}qxtn2` places the results to the highest half of the destination register where the lowest half is untouched. `{u, s}xtl{, 2}` extracts elements of the lowest or the highest half and unsigned-extends or signed-extends them to double-size elements. See Table 8.14 for a summary of the operands of widening and narrowing operations.

Table 8.14: Summary of operands of widening and narrowing operations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`xtn`	8B/4H/2S	8H/4S/2D
`xtn2`	16B/8H/4S	8H/4S/2D
`{u, s}qxtn`	B/H/S	H/S/D
`{u, s}qxtn`	8B/4H/2S	8H/4S/2D
`{u, s}qxtn2`	16B/8H/4S	8H/4S/2D
`{u, s}xtl`	8H/4S/2D	8H/4S/2D
`{u, s}xtl2`	8H/4S/2D	16H/8S/4D

Table 8.15: Summary of operands of comparisons in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`cmeq`	B/H/S/D	B/H/S/D
`cmeq`	Vector	Vector
`cm{ge, gt, hs, hi}`	B/H/S/D	B/H/S/D
`cm{ge, gt, hs, hi}`	Vector	Vector
`cm{le, lt}`	B/H/S/D	B/H/S/D, 0
`cm{le, lt}`	Vector	Vector, 0

Comparisons. For comparisons, `cmeq` tests if elements of the two source registers are pair-wise equal, `cmge` tests if elements of the first source register are greater than or equal to the ones in the second source register as signed integers, `cmgt` tests if elements of the first are greater than the ones in the second as signed integers, `cmle` tests if elements of the first are less than or equal to zeros as signed integers, `cmlt` tests if elements of the first are less than zeros as signed integers, `cmhs` tests if elements of the first are greater than or equal to the ones in the second as unsigned integers, and `cmhi` tests if elements of the first are greater than the ones in the second as unsigned integers [ARM21a, Table C3-81]. See Table 8.15 for a summary of operands of comparisons.

Bitwise operations. For bitwise operations, `not/mvn` complements the source register, `and` computes the bitwise and of the two source registers, `eor` computes the bitwise exclusive-or of the two source registers, `orr` computes the bitwise or of the two source registers, `orn` computes the bitwise or of the two source registers where the second source register is complemented, `bic` computes the bitwise and of the two source registers where the second source register is complemented, and `bsl` selects the desired bits from the two source registers [ARM21a, Tables C3-80 and C3-83]. There are three operands in `bsl`: the first register selects the desired of bits the rest of the two, and the results are destructively written to the first register. See Table 8.16 for a summary of operands of bitwise operations.

Table 8.16: Summary of operands of bitwise operations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`not/mvn/eor/orr/orn/and/bic/bsl`	8B/16B	8B/16B
`bic`	8B/16B	imm8

Bit-field operations. `bit` bitwisely inserts the bits of the first source register into the destination register if the corresponding bit of the second source register is 1, and `bif` inserts the bits if the corresponding bits of the second source register is 0 [ARM21a, Table C3-80]. See Table 8.17 for a summary of operands of bit-field insertions.

Table 8.17: Summary of operands of bit-field operations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`bit/bif`	8B/16B	8B/16B

Counting operations. For counting operations, `cls` counts the number of leading sign bits where the most significant bits are excluded, `clz` counts the number of leading zeros, and `cnt` counts the number of ones in each byte [ARM21a, Table C3-83]. See Table 8.18 for a summary of operands of counting operations.

Table 8.18: Summary of operands of counting operations in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`cls/clz`	8B/16B/4H/8H/2S/4S	8B/16B/4H/8H/2S/4S
`cnt`	8B/16B	8B/16B

Table 8.19: Summary of operands of right-shifts in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`{u, s}{, r}{shr, sra}`	D	D, `imm6`
`{u, s}{, r}{shr, sra}`	Vector	Vector, `imm6`
`sri`	D	D, `imm6`
`sri`	Vector	Vector, `imm6`
`{, r}shrn`	8B/4H/2S	8H/4S/2D, `imm4`
`{, r}shrn2`	16B/8H/4S	8H/4S/2D, `imm4`
`sq{, r}shr{, u}n`	B/H/S	H/S/D
`sq{, r}shr{, u}n`	8B/4H/2S	8H/4S/2D
`sq{, r}shr{, u}n2`	16B/8H/4S	8H/4S/2D
`uq{, r}shrn`	B/H/S	H/S/D
`uq{, r}shrn`	8B/4H/2S	8H/4S/2D
`uq{, r}shrn2`	16B/8H/4S	8H/4S/2D

Right shifts. For right shifts, `{u, s}{, r}shr` right-shifts in zeros or sign bits with optional rounding, and `{u, s}{, r}sra` accumulates the results to accumulators. `sri` right-shifts in zeros and inserts the results into the destination where zeros shifted in are ignored. `{, r}shrn{, 2}` right-shifts in zeros with optional rounding, and places the lowest halve-sized elements to the lowest or the highest half of the destination register. `sq{, r}shrn{, 2}` right-shifts in sign bits with optional rounding, saturates the results to half-sized elements as signed integers, and place the results to the lowest or the highest half of the destination register. `sq{, r}shrun{, 2}` right-shifts in sign bits with optional rounding and saturates the results to half-sized elements as unsigned integers. `uq{, r}shrn{, 2}` right-shifts in zeros with optional rounding and saturates the results to half-sized elements as signed integers. See Table 8.19 for a summary of right-shifts.

Left shifts. For left shifts, `shl` left-shifts in zeros and `sli` additionally inserts the results into the destination register where the zeros shifted in are ignored. `{u, s}{, r}shl` left-shifts in zeros for positive shift counts, and right-shifts in zeros or sign bits with optional rounding for negative shift counts. `{u, s}{q, qr}shl` performs the same operations as `{u, s}{, r}shl` except that all the intermediate results are saturated. `sqshlu` left-shifts in zeros and saturates the results as unsigned integers. `shll{, 2}` extracts the lowest or the highest half of the source register, extends the elements to double-sized ones, and left-shifts in zeros by element size. `{u, s}shll{, 2}` extracts the lowest or the highest half of the source register, extends the elements to double-sized ones as unsigned or signed integers, and left-shifts in zeros. See Table 8.20 for a summary of operands of left-shifts.

Table 8.20: Summary of operands of left-shifts in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`shl/sli`	`D`	`D`, `imm6`
`shl/sli`	Vector	Vector, `imm6`
`{u, s}{, r, qr}shl`	`D`	`D`
`{u, s}{, r, qr}shl`	Vector	Vector
`{u, s}qshl`	`D`	`D`, `imm6`
`{u, s}qshl`	Vector	Vector, `imm6`
`sqshlu`	`D`	`D`, `imm6`
`sqshlu`	Vector	Vector, `imm6`
`shll`	`8H/4S/2D`	`8B/4H/2S`, `8/16/32`
`shll2`	`8H/4S/2D`	`16B/8H/4S`, `8/16/32`
`{u, s}shll`	`8H/4S/2D`	`8B/4H/2S`
`{u, s}shll2`	`8H/4S/2D`	`16B/8H/4S`

Table 8.21: Summary of operands of multiplications in Armv8-A Neon [ARM21a].

Instruction	Destination	Source
`mul`	`8B/16B/4H/8H/2S/4S`	`8B/16B/4H/8H/2S/4S`
`mul`	`4H/8H/2S/4S`	`4H/8H/2S/4S`, `H[]/S[]`
`mla`/`mls`	`4H/8H/2S/4S`	`4H/8H/2S/4S`
`mla`/`mls`	`4H/8H/2S/4S`	`4H/8H/2S/4S`, `H[]/S[]`
`sq{, r}dmulh`	`H/S`	`H/S`, `H[]/S[]`
`sq{, r}dmulh`	`4H/8H/2S/4S`	`4H/8H/2S/4S`
`sqrd{mla, mls}h`	`H/S`	`H/S`, `H[]/S[]`
`sqrd{mla, mls}h`	`4H/8H/2S/4S`	`4H/8H/2S/4S`
`{u, s}mull`	`4S/2D`	`4H/2S`, `H[]/S[]`
`{u, s}mull`	`4S/2D`	`4H/2S`
`{u, s}mull2`	`4S/2D`	`8H/4S`, `H[]/S[]`
`{u, s}mull2`	`4S/2D`	`8H/4S`
`{u, s}mlal`	`4S/2D`	`4H/2S`, `H[]/S[]`
`{u, s}mlal`	`4S/2D`	`4H/2S`
`{u, s}mlal2`	`4S/2D`	`8H/4S`, `H[]/S[]`
`{u, s}mlal2`	`4S/2D`	`8H/4S`
`{u, s}mlsl`	`4S/2D`	`4H/2S`, `H[]/S[]`
`{u, s}mlsl`	`4S/2D`	`4H/2S`
`{u, s}mlsl2`	`4S/2D`	`8H/4S`, `H[]/S[]`
`{u, s}mlsl2`	`4S/2D`	`8H/4S`
`sqdmull`	`S/D`	`H/S`, `H[]/S[]`
`sqdmull`	`4S/2D`	`4H/2S`
`sqdmull2`	`4S/2D`	`8H/4S`
`sqd{mla, mls}l`	`S/D`	`H/S`, `H[]/S[]`
`sqd{mla, mls}l`	`4S/2D`	`4H/2S`
`sqd{mla, mls}l2`	`4S/2D`	`8H/4S`

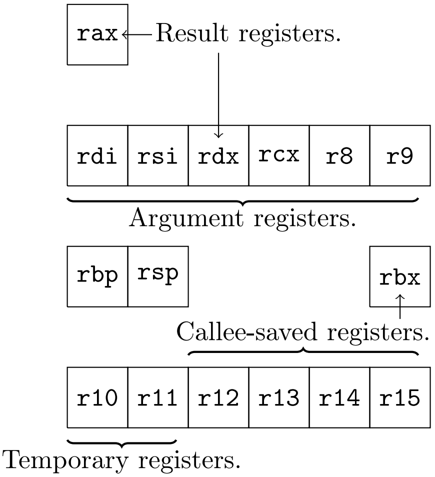
Multiplications. For multiplications, we have the standard ones `mul`, `mla`, `mls` [ARM21a, Table C3-80] and several ones computing the high parts and the full size products. `sq{, r}dmulh` computes the products of the corresponding elements of the two source registers as signed integers, doubles the products with saturation, and extracts the highest halves with optional rounding [ARM21a, Table C3-80]. `sqrd{mla, mls}h` additionally accumulates or subtracts the results from the accumulators [ARM21a, Table C3-80]. `{u, s}{mul,` `mla, mls}l{, 2}` computes the full size products of the lowest or the highest halves as unsigned or signed integers and optionally accumulates or subtracts the results from the accumulators [ARM21a, Table C3-82]. `sqd{mul, mla, mls}l{, 2}` doubles the products, saturates the them, and optionally accumulates or subtracts the results from the accumulators [ARM21a, Table C3-82]. See Table 8.21 for a summary of operands of multiplications.

8.1.3 AVX2

x86-64 is a 64-bit ISA extension implemented in personal computers and servers. There are several vector extensions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, and AVX-512. This thesis focuses on the AVX2 vector extension.

8.1.3.1 General Purpose Registers

Figure 8.6: General-purpose registers in x86-64 [AMD25]. Each rectangle represents a 64-bit register.

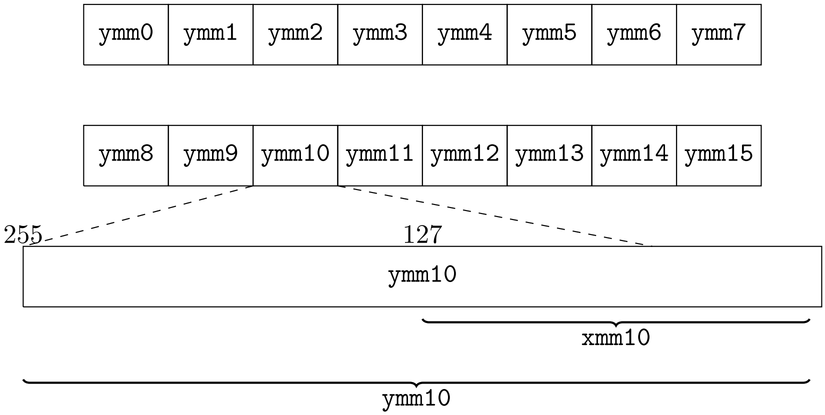


There are 16 64-bit general purpose register: `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `rbx`, `rbp`, `rsp`, `r10`, . . . , `r15`. We recall the register usage of System V Application Binary Interface [AMD25, Figure 3.4]. `rax` and `rdx` are result registers, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` are argument registers, `rbp` is the frame pointer, `rsp` is the stack pointer, `r10` and `r11` are temporary registers, and the rest are callee-saved registers. See Figure 8.6 for an illustration.

8.1.3.2 SIMD Registers

In AVX2, there are 16 256-bit SIMD registers: `ymm0`, . . . , `ymm15`. The lowest 128 bits of `ymm`'s are also named as `xmm`'s. See Figure 8.7 for an illustration.

Figure 8.7: SIMD registers in AVX2. Each rectangle represents a 256-bit register.



8.1.3.3 AVX2 Instructions

Permutations. For permutations, `vpunpck{l, h}bw` interleaves the 8-bit values in the lowest or the highest half of each 128-bit value, `vpunpck{l, h}wd` interleaves the 16-bit values in the lowest or the highest half of each 128-bit value, `vpunpck{l, h}dq` interleaves the 32-bit values in the lowest or the highest half of each 128-bit value, and `vpunpck{l, h}qdq` interleaves the 64-bit values in the lowest or the highest half of each 128-bit value. `vperm2i128` selects two 128-bit values from the two source registers and concatenates them. See Table 8.22 for a summary of operands of permutations.

Table 8.22: Summary of operands of permutations in AVX2.

Instruction	Destination	Source
`vpunpck{l, h}bw`	32 × 8-bit	32 × 8-bit
`vpunpck{l, h}wd`	16 × 16-bit	16 × 16-bit
`vpunpck{l, h}dq`	8 × 32-bit	8 × 32-bit
`vpunpck{l, h}qdq`	4 × 64-bit	4 × 64-bit
`vperm2i128`	2 × 128-bit	2 × 128-bit

Additions and subtractions. For additions and subtractions, `vp{add, sub}{b, w, d, q}` adds or subtracts 8-bit, 16-bit, 32-bit, or 64-bit values, `vp{add, sub}{, u}s{b, w}` adds or subtracts 8-bit or 16-bit values with signed or unsigned saturation, `vph{add, sub}{w, d}` interleaves each 128-bit of the two source registers and horizontally adds or subtracts 16-bit or 32-bit values, `vph{add, sub}sw` interleaves each 128-bit of the two source registers and horizontally adds or subtracts 16-bit values with saturation. See Table 8.23 for a summary of operands.

Table 8.23: Summary of operands of additions and subtractions in AVX2.

Instruction	Destination	Source
`vp{add, sub}{, s, us}b`	32 × 8-bit	32 × 8-bit
`vp{add, sub}{, s, us}w`	16 × 16-bit	16 × 16-bit
`vp{add, sub}d`	8 × 32-bit	8 × 32-bit
`vp{add, sub}q`	4 × 64-bit	4 × 64-bit
`vph{add, sub}{, s}w`	16 × 16-bit	16 × 16-bit
`vph{add, sub}d`	8 × 32-bit	8 × 32-bit

Bitwise operations. For bitwise operations, `vpand{, n}` optionally negates the second source register and computes the bitwise and, `vpor` computes the bitwise or, and `vpxor` computes the bitwise exclusive-or. See Table 8.24 for a summary of operands.

Table 8.24: Summary of operands of bitwise operations in AVX2.

Instruction	Destination	Source
`vpand{, n}`	1 × 256-bit	1 × 256-bit
`vp{, x}or`	1 × 256-bit	1 × 256-bit

Shifts. For shifts, `vps{l, r}l{w, d, q}` left-shifts or right-shifts 16-bit, 32-bit, or 64-bit values with zeros, `vps{l, r}lv{d, q}` variably left-shifts or right-shifts 32-bit or 64-bit values with zeros, `vpsra{w, d, q}` right-shifts 16-bit, 32-bit, or 64-bit values with sign bits, and `vpsravd` variably right-shifts 32-bit values with sign bits. `vps{l, r}ldq` left-shifts or right-shifts each 128-bit values by the specified number of bytes with zeros. See Table 8.25 for a summary of operands.

Table 8.25: Summary of operands of shifts in AVX2.

Instruction	Destination	Source
`vps{l, r}lw`	16 × 16-bit	16 × 16-bit, `imm8`
`vps{l, r}lw`	16 × 16-bit	16 × 16-bit, 2 × 64-bit
`vps{l, r}ld`	8 × 32-bit	8 × 32-bit, `imm8`
`vps{l, r}ld`	8 × 32-bit	8 × 32-bit, 2 × 64-bit
`vps{l, r}lq`	4 × 64-bit	4 × 64-bit, `imm8`
`vps{l, r}lq`	4 × 64-bit	4 × 64-bit, 2 × 64-bit
`vps{l, r}lvd`	4 × 32-bit	4 × 32-bit
`vps{l, r}lvd`	8 × 32-bit	8 × 32-bit
`vps{l, r}lvq`	2 × 64-bit	2 × 64-bit
`vps{l, r}lvq`	4 × 64-bit	4 × 64-bit
`vpsraw`	16 × 16-bit	16 × 16-bit, `imm8`
`vpsraw`	16 × 16-bit	16 × 16-bit, 2 × 64-bit
`vpsrad`	8 × 32-bit	8 × 32-bit, `imm8`
`vpsrad`	8 × 32-bit	8 × 32-bit, 2 × 64-bit
`vpsravd`	4 × 32-bit	4 × 32-bit
`vps{l, r}ldq`	2 × 128-bit	2 × 128-bit, `imm8`

Multiplications. For multiplications, `vpmaddubsw` multiplies unsigned 8-bit values with signed 8-bit values, horizontally adds adjacent 16-bit results as signed integers, and saturates the results, and `vpmaddwd` multiplies signed 16-

bit values and horizontally adds adjacent 32-bit results. `vpmul{l, h}w` multiplies signed 16-bit values and extracts the lowest or the highest 16-bit values, `vpmulhuw` multiplies unsigned 16-bit values and extracts the highest 16-bit values, `vpmulhrsw` multiplies signed 16-bit values, rounds to the highest 17-bit values, and drops the highest bits. `vpmulld` multiplies 32-bit values, and `vpmul{d, ud}q` computes the 64-bit products of the lowest signed or unsigned 32-bit values. See Table 8.26 for a summary of operands.

Table 8.26: Summary of operands of multiplications in AVX2.

Instruction	Destination	Source
`vpmaddubsw`	16 × 16-bit	32 × 8-bit
`vpmaddwd`	8 × 32-bit	16 × 16-bit
`vpmul{l, h}w`	16 × 16-bit	16 × 16-bit
`vpmulh{u, rs}w`	16 × 16-bit	16 × 16-bit
`vpmulld`	8 × 32-bit	8 × 32-bit
`vpmul{d, ud}q`	8 × 32-bit	4 × 64-bit

8.2 Processors

This section reviews the processors covered by this thesis. A program consists of a string of assembly instructions and is sent to a processor when we execute it. The processor first decodes the assembly instructions of the program into a string of micro-operations and dispatches the micro-operations to the backend computation units.

Single-issue processors. On a single-issue processor, assembly instructions are usually decoded one at a time and micro-operations are executed one at a time. This section reviews the cycles of instructions on single-issue processors.

Superscalar processors. On a superscalar processor, several assembly instructions are decoded at once, and several micro-operations are dispatched to the execution units and executed at once when possible. The throughput of an instruction is defined as the maximum number of the same instructions that can be executed at the same time. A common way to evaluate the throughput of an instruction is to measure the cycles of a string of same instructions with independent operands. The inverse throughput (IT) of an instruction is defined as the inverse of the throughput and translates into the averages cycles of the instruction. In a large program, inverse throughput roughly translates into the lower bound of the cycles of a program, and provides a reasonable way to evaluate how well the approach is. This section reviews the inverse throughput of instructions on superscalar processors.

8.2.1 Cortex-M3

Cortex-M3 implements the 32-bit Armv7-M. We recall below the instruction timings of load, store, and arithmetic instructions.

Load/store instruction timings. A single-register store with an immediate offset is always pipelined and takes one cycle. A single-register store with register offset can only be pipelined after a load, does not pipeline with the follow-up instruction, and takes two cycles. Two-register and multi-register load and store do not pipeline with other instructions and take $n+2$ cycles to load or store n registers. As for single-register load, the base-updating ones do not pipeline with register-read instructions and take two cycles in general. As for the non-base-updating single-register load, if the next instruction is a non-base-updating single-register store or a single-register load with independent base address, then it is pipelined with the next instruction and takes one cycle. Otherwise, it takes two cycles [ARM10a, Section 18.3].

Arithmetic instruction timing. Excluding PC as an operand, each bitfield operation, bitwise operation, shift, byte-level permutation, addition, subtraction takes one cycles. `mul` takes one cycle, and `mla`/`mls` takes two cycles. As for long multiplications `{u, s}{mul, mla}l`, each takes 3 to 7 cycles depending on the absolute value of the result. See Table 8.27 for a summary.

Instruction	Cycle
`add`/`adc`/`sub`/`sbc`/`rsb`/`neg`/ `mov`/`and`/`bic`/`orr`/`orn`/`eor`/ `bfc`/`bfi`/`uxtb`/`uxth`/`ubfx`/`sxtb`/`sxth`/`sbfx`/ `lsl`/`lsr`/`asr`/`ror`/ `rev`/`revh`/`rev16`/ `mul`	1
`mla`/`mls`	2
`smull`/`smlal`/`umull`/`umlal`	3–7

Table 8.27: Summary of arithmetic instruction timings on Cortex-M3.

8.2.2 Cortex-M4

Cortex-M4 implements the 32-bit ISA Armv7E-M and comes with optional FPv4-SP extension. We recall below the instruction timings of load, store, arithmetic, and floating-point transferring instructions.

Load/store instruction timings. The load/store instructions behave the same as Cortex-M3 except that base-updating single-register loads are pipelined and take one cycle when possible [ARM10b, Section 3.3.2].

Arithmetic instruction timings. Excluding PC as an operand, each bit-field operation, bitwise operation, shift, byte-level permutation, addition, subtraction, multiplication takes 1 cycle [ARM10b, ARM10c].

Floating-point transferring instruction timings. For floating-point transferring operations, if the source or the destination consists of a single-precision floating-point register, then the operation takes one cycle; and if the source or the destination consists of two single-precision floating-point registers, then the operation takes two cycles.

8.2.3 Cortex-A72

Cortex-A72 implements the 64-bit Armv8-A. Below we recall the execution ports and instruction characteristics of the SIMD instructions reviewed in Section 8.1.2.2.

8.2.3.1 Execution Ports and Instruction Characteristics

On Cortex-A72, there are one branch execution port `B`, two integer execution ports `I0` and `I1`, one multi-cycle execution port `M`, one load execution port `L`, one store execution port `S`, and two floating-point/SIMD (FP/SIMD) execution ports `F0` and `F1` [ARM15, Section 2.1].

8.2.3.2 Instruction Characteristics

We recall the instruction characteristics of the SIMD instructions reviewed in Section 8.1.2.2.

Loads and stores. Loading 8-bit, 16-bit, 32-bit, 64-bit, 128-bit elements with `ldr` goes to port `L` has inverse throughput 1 [ARM15, Section 3.9]. Storing 8-bit, 16-bit, 32-bit, 64-bit, 128-bit elements with `str` goes to port `S` and has inverse throughput 1 with 8-bit, 16-bit, 32-bit, 64-bit source register and inverse throughput 0.5 with 128-bit source register. The base-updating versions also occupy one of the ports `I0` and `I1` [ARM15, Section 3.9].

Transferring operations. Duplicating a lane from a SIMD register to a SIMD register with `dup` goes to one of the ports `F0` and `F1` and has inverse throughput 2; duplicating an element from a general purpose register to a SIMD register with `dup` goes to port `L` and one of the ports `F0` and `F1` and has inverse throughput 1; moving an 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit element from a SIMD register to a SIMD register with `mov` goes to one of the ports `F0` and `F1` and has inverse throughput 2 [ARM15, Sections 3.14 and 3.16]. The inverse throughput of moving an 8-bit, 16-bit, 32-bit element from a SIMD register to another one is missing from [ARM15] and standalone benchmarks are included in the artifact.

Permutations. Each of the permutations reviewed in Section 8.1.2.3 goes to one of the ports `F0` and `F1` and its inverse throughput is 2 [ARM15, Section 3.16].

Widening and narrowing operations. `xtn{, 2}` goes to one of the ports `F0` and `F1` and has inverse throughput 2, and `{u, s}qxtn{, 2}` and `{u, s}xtl{, 2}` goes to port `F1` and has inverse throughput 1 [ARM15, Section 3.16]. The inverse throughput of `xtn2` is missing from [ARM15] and standalone benchmarks are included in the artifact.

Additions and subtractions. Each of the additions and subtractions reviewed in Section 8.1.2.3 goes to one of the ports `F0` and `F1` and has inverse throughput 2 [ARM15, Section 3.14].

Comparisons. Each of the comparisons reviewed in Section 8.1.2.3 goes to one of the ports `F0` and `F1` and has inverse throughput 2 [ARM15, Section 3.14].

Bitwise operations. Each of the bitwise operations reviewed in Section 8.1.2.3 goes to one of the ports `F0` and `F1` and has inverse throughput 2 [ARM15, Section 3.14].

Right shifts. Excluding `sri`, each of the right shifts reviewed in Section 8.1.2.3 goes to port `F1` and has inverse throughput 1 [ARM15, Section 3.14]. `sri` goes to port `F1` and its inverse throughput is 1 with a 64-bit source register and 0.5 with a 128-bit source register [ARM15, Section 3.14]. `sri` is listed as a "shift by immed" and a ""shift by immed and insert" in [ARM15]. This thesis confirms the latter and standalone benchmarks are included in the artifact.

Left shifts. Excluding `sli`, each of the left shifts reviewed in Section 8.1.2.3 goes to port `F1` and has inverse throughput 1 with an immediate shift count, and inverse throughput 0.5 with a register holding the shift counts [ARM15, Section 3.14]. `sli` goes to port `F1` and its inverse throughput is 1 with a 64-bit source register and 2 with a 128-bit source register [ARM15, Section 3.14].

Multiplications. Excluding the widening multiplications, each of the multiplications reviewed in Section 8.1.2.3 goes to port `F0` and has inverse throughput 1 with a 64-bit source register and inverse throughput 0.5 with a 128-bit source register [ARM15, Section 3.14]. Each widening multiplication goes to port `F0` and has inverse throughput 1 [ARM15, Section 3.14].

8.2.4 Firestorm

Firestorm also implements the 64-bit Armv8-A. Below we recall the execution ports and instruction characteristics of the SIMD instructions reviewed in Section 8.1.2.2. See [Joh] for a complete report.

8.2.4.1 Execution Ports and Instruction Characteristics

On a Firestorm, there are six integer execution ports, one store execution port, two load execution ports, one load/store execution port, and four floating-point/SIMD execution ports.

8.2.4.2 Instruction Characteristics

Loads and stores. Each non-base-updating load has inverse throughput 0.333 and the base-updating ones are slightly slower. Each non-base-updating store has inverse throughput 0.5 and the base-updating ones are slightly slower.

Transferring operations. Duplicating an element from a general-purpose register to a SIMD register has inverse throughput 0.333, duplicating an element from a lane to a SIMD register has inverse throughput 0.25, transferring an element from a general-purpose register to a lane of a SIMD register has inverse throughput 0.333, transferring an element from a lane of a SIMD register to a general-purpose register has inverse throughput 0.5, and transferring an element from a lane of a SIMD register to a lane of another SIMD register has inverse throughput 0.25. Transferring the whole content of a SIMD register to another SIMD register has inverse throughput 0.125.

Arithmetic operations. Each of the SIMD permutations, widening operations, narrowing operations, additions, subtractions, comparisons, bitwise operations, bit-field operations, counting operations, right shifts, left shifts, and multiplications has inverse throughput 0.25.

8.2.5 Haswell

Haswell implements the x86-64 with AVX2. Below we recall the execution ports and instruction characteristics from [Fog].

8.2.5.1 Execution Ports and Instruction Characteristics

On Haswell, there are eight execution ports: two load execution ports `p2` and `p3`, one store execution port `p4`, one store address execution port `p7`, and four integer arithmetic execution ports `p0`, `p1`, `p5`, and `p6`. Among the four integer arithmetic execution ports, the three execution ports `p0`, `p1`, and `p5` are capable of vector arithmetic. Among the three execution ports capable of vector arithmetic, the two execution ports `p0` and `p1` are capable of floating-point arithmetic. Among the two execution ports capable of floating-point arithmetic, execution port `p0` is capable of vector multiplications.

8.2.5.2 Instruction Characteristics

Permutations. Each permutations goes to `p5` and has inverse throughput 1.

Additions and subtractions. `vp{add, sub}{b, w, d, q}` and `vp{add, sub}{, u}s{b, w}` goes to one of `p1` and `p5` and has inverse throughput 0.5. As for `vph{add, sub}{w, d}` and `vph{add, sub}sw`, each is decoded into one micro-operation going to `p1` and two micro-operations going to `p5` and has inverse throughput 2.

Bitwise operations. Each bitwise operations goes to one of `p0`, `p1`, and `p5` and has inverse throughput 0.33.

Shifts. For shifts, each `vps{ll, rl, ra}{w, d, q}` with `imm8` shift count goes to `p0` and has inverse throughput 1, and each `vps{ll, rl, ra}{w, d, q}` with `xmm` shift counts is decoded into one micro-operation going to `p0` and one micro-operation going to `p5` and has inverse throughput 1. Each `vps{ll, rl}v{d, q}` and `vpsravd` is decoded into two micro-operations going to `p0` and one micro-operation going to `p5` and has inverse throughput 2. Each `vpsl{l, r}dq` goes to `p5` and has inverse throughput 1.

Multiplications. For multiplications, each multiplications except for `vpmulld` goes to `p0` and has inverse throughput 1, and `vpmulld` is decoded into two micro-operations going to `p0` and has inverse throughput 2.

Chapter 9

Implementations of Modular Multiplications (Mulmod.) and Quotients

This chapter details the implementations of modular multiplications and quotient computations. Section 9.1 goes through the multiplications instructions in Armv7-M, Armv7E-M, Armv8-A Neon, and AVX2, Section 9.2 goes through the modular multiplications, including Barrett, Montgomery, and Plantard multiplications, using multiplication instructions from Section 9.1, and Section 9.3 goes through the quotient computations based on the notion of modular multiplication and explicit constructions from Barrett multiplication. This chapter also gives the cycle count estimation based on the most heavily utilized pipeline of the processors.

9.1 Multiplications

We categorize multiplications into five categories: low multiplication, high multiplication, long multiplication, wide multiplication, inner product. Table 9.1 is an overview, and we will go through each ISA in the follow-up sections.

Low multiplications. A low multiplication computes lower $\log_2 \mathtt{R}$ bits of the product of two $\log_2 \mathtt{R}$-bit operands. The accumulative variant accumulates the result to a $\log_2 \mathtt{R}$-bit accumulator, and the subtractive variant subtracts the result from the accumulator.

Table 9.1: Overview of the available multiplications in Armv7-M, Armv7E-M, Armv8 Neon, and AVX2.

ISA/Extension	Base	Accumulative	Subtractive
Low multiplication			
Armv7-M	$\mathtt{R} = 2^{32}$	$\mathtt{R} = 2^{32}$	$\mathtt{R} = 2^{32}$
Armv7E-M	$\mathtt{R} = 2^{32}$	$\mathtt{R} = 2^{32}$	$\mathtt{R} = 2^{32}$
Armv8.0-A Neon	$\mathtt{R} = 2^8, 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$
AVX2	$\mathtt{R} = 2^{16}, 2^{32}$	-	-
High multiplication			
Armv7-M	-	-	-
Armv7E-M	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{32}$
Armv8.0-A Neon	$\mathtt{R} = 2^{16}, 2^{32}$	-	-
Armv8.1-A Neon	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$
AVX2	$\mathtt{R} = 2^{16}$	-	-
Long multiplication			
Armv7-M	$\mathtt{R} = 2^{32}$	$\mathtt{R} = 2^{32}$	-
Armv7E-M	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$	-
Armv8.0-A Neon	$\mathtt{R} = 2^8, 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$	$\mathtt{R} = 2^{16}, 2^{32}$
AVX2	$\mathtt{R} = 2^{32}$	-	-
Wide multiplication			
Armv7-M	-	-	-
Armv7E-M	$\mathtt{R} = 2^{16}$	$\mathtt{R} = 2^{16}$	-
Armv8-A Neon	-	-	-
AVX2	-	-	-
Inner product			
Armv7-M	-	-	-
Armv7E-M	$\mathtt{R} = 2^{16}$	$\mathtt{R} = 2^{16}$	-
Armv8.4-A Neon	-	$\mathtt{R} = 2^8$	
AVX2	$\mathtt{R} = 2^8, 2^{16}$	-	-

High multiplications. A high multiplication computes a reasonably accurate approximation of the upper $\log_2 \mathtt{R}$ bits of the product of two $\log_2 \mathtt{R}$-bit operands. The accumulative variant accumulates the result to a $\log_2 \mathtt{R}$-bit accumulator, and the subtractive variant subtracts the result from the accumulator.

Long multiplications. A long multiplication computes the full $2 \log_2 \mathtt{R}$-bit product of two $\log_2 \mathtt{R}$-bit operands. The accumulative variant accumulates the result to a $2 \log_2 \mathtt{R}$-bit accumulator, and the subtractive variant subtracts the result from the accumulator.

Wide multiplications. A wide multiplication computes a reasonably accurate approximation of the upper $2 \log_2 \mathtt{R}$ bits of the product of a $\log_2 \mathtt{R}$-bit operand and a $2 \log_2 \mathtt{R}$-bit operand. The accumulative variant accumulates the result to a $2 \log_2 \mathtt{R}$-bit accumulator, and the subtractive variant subtracts the result from the accumulator.

Inner products. An inner product computes several $2 \log_2 \mathtt{R}$-bit products from $\log_2 \mathtt{R}$-bit operands and sums up the products. The products are sometimes extended to $4 \log_2 \mathtt{R}$-bit elements before the summation. If there are only two pairs of $\log_2 \mathtt{R}$-bit operands, the summation is sometimes replaced by subtraction. The resulting sum of products are accumulated to the accumulator in the accumulative variant.

9.1.1 Armv7-M and Armv7E-M

In Armv7-M, we only have low multiplications `mul`, `mla`, and `mls`, and long multiplications `{u, s}{mul, mla, mls}l` with 32-bit general-purpose registers as inputs. As for multiplications in the DSP extension of Armv7E-M, each 32-bit register is also regarded as packed 16-bit elements. We have 32-bit high multiplications `sm{mul, mla, mls}{, r}`, 16-bit wide multiplications `s{mul, mla}w{b, t}`, 16-bit long multiplications `s{mul, mla, mls}{b, t}{b, t}`, `s{mla, mls}l{b, t}{b, t}`, and 16-bit inner products `s{mua, mus, mla, mls}d{, x}`, `s{mla, mls}ld{, x}`. See Table 9.2 for a summary.

Table 9.2: Multiplications in Armv7E-M.

Type	32-bit	16-bit (DSP)	32-bit (DSP)
Low multiplication			
Base	`mul`	-	-
Accumulative	`mla`	-	-
Subtractive	`mls`	-	-
High multiplication			
Base	-	-	`smmul{, r}`
Accumulative	-	-	`smmla{, r}`
Subtractive	-	-	`smmls{, r}`
Wide multiplication			
Base	-	`smulw{b, t}`	-
Accumulative	-	`smlaw{b, t}`	-
Long multiplication			
Base	`{u, s}mull`	`smul{, l}{b, t}{b, t}`	-
Accumulative	`{u, s}mlal`	`smla{, l}{b, t}{b, t}`	-
Subtractive	-	`smls{, l}{b, t}{b, t}`	-
Inner product			
Base	-	`s{mua, mus}d{, x}`	-
Accumulative	-	`s{mla, mls}{, l}d{, x}`	-

9.1.2 Armv8-A Neon

In Armv8-A Neon, each 128-bit SIMD register is regarded as packed 8-bit, 16-bit, 32-bit input elements. We have low multiplications `mul`, `mla`, `mls`, and `pmul`, high multiplications `sq{, r}mulh`, `sqrd{mla, mls}h`, long multiplications `{u, s}{mul, mla, mls}l{, 2}`, `sqd{mul, mla, mls}l{, 2}`, `pmull{, 2}`, and inner products `{u, s}dot`. `pmul`, `pmull{, 2}` compute products of binary polynomials and are out of the scope of this thesis, and `sqrd{mla, mls}h` only exist after Armv8.1-A. `{u, s}dot` are optional in Armv8.2-A and Armv8.3-A, mandatory after Armv8.4-A, and are out of the scope of this thesis. All other instructions are implemented in Armv8.0-A. See Table 9.3 for a summary.

Table 9.3: Multiplications in Armv8-A Neon.

Type	Instruction
Low multiplication	
Base	`mul` (16B/8H/4S), `pmul` (16B)
Accumulative	`mla` (8H/4S)
Subtractive	`mls` (8H/4S)
High multiplication	
Base	`sq{, r}dmulh` (8H/4S)
Accumulative	`sqrdmlah` (8H/4S)
Subtractive	`sqrdmlsh` (8H/4S)
Long multiplication	
Base	`{u, s}mull{, 2}` (4H/8H/2S/4S), `sqdmull{, 2}` (4H/8H/2S/4S), `pmull{, 2}` (8B/16B/1D/2D)
Accumulative	`{u, s}mlal{, 2}` (4H/8H/2S/4S), `sqdmlal{, 2}` (4H/8H/2S/4S)
Subtractive	`{u, s}mlsl{, 2}` (4H/8H/2S/4S), `sqdmlsl{, 2}` (4H/8H/2S/4S)
Inner product	
Accumulative	`{u, s}dot` (8B/16B)

9.1.3 AVX2

In AVX2, we have low multiplications `vpmull{w, d}`, high multiplications `vpmul{, u, rs}w`, long multiplications `vpmul{, u}dq`, and inner products `vpmadd{ubsw, wd}`. See Table 9.4 for a summary.

Table 9.4: Multiplications in AVX2.

Type	Instruction
Low multiplication	`vpmull{w, d}`
High multiplication	`vpmulh{, u, rs}w`
Long multiplication	`vpmul{, u}dq`
Inner product	`vpmadd{ubsw, wd}`

9.2 Modular Multiplications

9.2.1 On the Multiplication Instructions for Modular Multiplications

Before going through the implementations, we first dig into more details on the types of multiplication instructions used in modular multiplications.

9.2.1.1 Montgomery Multiplication

The classical one is Montgomery reduction with long multiplication defined as follows.

Definition 27 (Montgomery reduction with long multiplication)**.** Let q and $\mathtt{R}$ be coprime integers greater than 1. For an integer $a \in \left[-\frac{\mathtt{R}^2}{2}, \frac{\mathtt{R}^2}{2}\right) \cap \mathbb{Z}$, **the Montgomery reduction with long multiplication** computes

$$\frac{a + \left(a \left(-q^{-1} \bmod^{\pm} \mathtt{R}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}$$

as a representative of $a\mathtt{R}^{-1} \bmod^{\pm} q$ with long multiplication for the multiplication by q.

Theorem 8. Let q and $\mathtt{R}$ be coprime integers greater than 1. For an integer a, we have

$$\frac{a + \left(a \left(-q^{-1} \bmod^{\pm} \mathtt{R}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}} \leq \frac{|a|}{\mathtt{R}} + \frac{q}{2}$$

for the Montgomery reduction with long multiplication.

As for the Montgomery multiplication, we have the accumulative Montgomery with long multiplications, the subtractive Montgomery multiplication with long multiplications, and the subtractive Montgomery multiplication with high multiplications defined as follows.

Definition 28 (Accumulative Montgomery multiplication with long multiplications)**.** Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, **the accumulative Montgomery multiplication with long multiplications** computes

$$\frac{ab + \left(ab \left(-q^{-1} \bmod^{\pm} \mathtt{R}\right) \bmod^{\pm} \mathtt{R}\right) q}{\mathtt{R}}$$

as a representative of $ab\mathtt{R}^{-1} \bmod^{\pm} q$ with long multiplications for ab and the multiplication by q.

Definition 29 (Subtractive Montgomery multiplication with long multiplications)**.** Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, **the subtractive Montgomery multiplication with long multiplications** computes

$$\frac{ab - \left(ab\left(q^{-1} \bmod^{\pm}\mathtt{R}\right) \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}}$$

as a representative of $ab\mathtt{R}^{-1} \bmod^{\pm} q$ with long multiplications for ab and the multiplication by q.

Definition 30 (Subtractive Montgomery multiplication with high multiplications)**.** Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, **the subtractive Montgomery multiplication with high multiplications** computes

$$\left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(ab\left(q^{-1} \bmod^{\pm}\mathtt{R}\right) \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} \right\rfloor$$

as a representative of $ab\mathtt{R}^{-1} \bmod^{\pm} q$ with high multiplications for $\left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor$ and the multiplication by q.

Theorem 9. Let q and $\mathtt{R}$ be coprime integers greater than 1. For integers a, b, we have

$$\begin{cases} \frac{ab + \left(ab\left(-q^{-1} \bmod^{\pm}\mathtt{R}\right) \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} & \leq \frac{|ab|}{\mathtt{R}} + \frac{q}{2}, \\ \frac{ab - \left(ab\left(q^{-1} \bmod^{\pm}\mathtt{R}\right) \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} & \leq \frac{|ab|}{\mathtt{R}} + \frac{q}{2}, \\ \left\lfloor \frac{ab}{\mathtt{R}} \right\rfloor - \left\lfloor \frac{\left(ab\left(q^{-1} \bmod^{\pm}\mathtt{R}\right) \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} \right\rfloor & \leq \frac{|ab|}{\mathtt{R}} + \frac{q}{2} \end{cases}$$

for the accumulative Montgomery multiplication with long multiplications, the subtractive Montgomery multiplication with long multiplications, and the subtractive Montgomery multiplication with high multiplications.

9.2.1.2 Barrett Multiplication

For Barrett multiplication, we outline below four variants distinguished by the choice of the integer approximation for the quotient. See Table 9.5 for a summary.

Table 9.5: Overview of the variants of Barrett multiplications. Upper bounds stand for the upper bounds of the absolute values of the results.

Variant	Upper bound	Upper bound when $\lvert a\rvert \leq \frac{\mathtt{R}}{2}$
Standard	$\frac{q}{2}\left(1 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$0.75q$
Floor	$\frac{q}{2}\left(3 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$1.75q$
Half-approximate	$\frac{q}{2}\left(5 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$2.75q$
Approximate	$\frac{q}{2}\left(7 + \frac{\lvert a\rvert}{\mathtt{R}}\right)$	$3.75q$

Definition 31 (Standard Barrett multiplication with high multiplication)**.** Let q and $\mathtt{R}$ be integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, the **standard Barrett multiplication with high multiplication** computes

$$ab - \left\lfloor \frac{a\left\lfloor b\mathtt{R}/q\right\rfloor}{\mathtt{R}} \right\rceil q$$

as a representative of $ab \bmod^{\pm} q$ with high multiplication for the multiplication by the precomputed constant $\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$.

Definition 32 (Floor variant of Barrett multiplication with high multiplication)**.** Let q and $\mathtt{R}$ be integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, the **floor variant of Barrett multiplication with high multiplication** computes

$$ab - \left\lfloor \frac{a\left\lfloor b\mathtt{R}/q\right\rfloor}{\mathtt{R}} \right\rfloor q$$

as a representative of $ab \bmod^{\pm} q$ with high multiplication for the multiplication by the precomputed constant $\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$.

Definition 33 (Half-approximate variant of Barrett multiplication with high multiplication)**.** Let q and $\mathtt{R}$ be integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, and ${}_b[\![]\!]$ the following integer approximation:

$$\forall r \in \mathbb{R}, {}_b[\![r]\!] = \left\lfloor \frac{a_l b_h + \sqrt{\mathtt{R}}/2}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor + a_h b_h$$

where $a_l + a_h\sqrt{\mathtt{R}} = \frac{r\mathtt{R}}{\lfloor b\mathtt{R}/q\rfloor}, b_l + b_h\sqrt{\mathtt{R}} = \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$, and $a_l, b_l \in \left[0, \sqrt{\mathtt{R}}\right)$, the **half-approximate variant of Barrett multiplication with high multiplication** computes

$$ab -_b \left[\!\!\left[\frac{a\left\lfloor b\mathtt{R}/q\right\rfloor}{\mathtt{R}} \right]\!\!\right] q$$

as a representative of $ab \bmod^{\pm} q$ with high multiplication for the multiplication by the precomputed constant $\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$.

Definition 34 (Approximate variant of Barrett multiplication with high multiplication)**.** Let q and $\mathtt{R}$ be integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, and $\llbracket\rrbracket_b$ the following integer approximation:

$$\forall r \in \mathbb{R}, \llbracket r \rrbracket_b = \left\lfloor \frac{a_l b_h}{\sqrt{\mathtt{R}}} \right\rfloor + \left\lfloor \frac{a_h b_l}{\sqrt{\mathtt{R}}} \right\rfloor + a_h b_h$$

where $a_l + a_h\sqrt{\mathtt{R}} = \frac{r\mathtt{R}}{\lfloor b\mathtt{R}/q \rfloor}$, $b_l + b_h\sqrt{\mathtt{R}} = \left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$, and $a_l, b_l \in \left[0, \sqrt{\mathtt{R}}\right)$, the **approximate variant of Barrett multiplication with high multiplication** computes

$$ab - \left\llbracket \frac{a \lfloor b\mathtt{R}/q \rfloor}{\mathtt{R}} \right\rrbracket_b q$$

as a representative of $ab \bmod^{\pm} q$ with high multiplication for the multiplication by the precomputed constant $\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil$.

As for the output range of Barrett multiplications, we have the following.

Theorem 10. Let q and $\mathtt{R}$ be integers greater than 1. For integers $a, b \in \left[-\frac{\mathtt{R}}{2}, \frac{\mathtt{R}}{2}\right) \cap \mathbb{Z}$, we have

$$\begin{cases} \left| ab - \left\lfloor \frac{a \lfloor b\mathtt{R}/q \rfloor}{\mathtt{R}} \right\rceil q \right| & \leq \frac{q}{2}\left(1 + \frac{|a|}{\mathtt{R}}\right), \\ \left| ab - \left\lfloor \frac{a \lfloor b\mathtt{R}/q \rfloor}{\mathtt{R}} \right\rfloor q \right| & \leq \frac{q}{2}\left(3 + \frac{|a|}{\mathtt{R}}\right), \\ \left| ab -_b \left\llbracket \frac{a \lfloor b\mathtt{R}/q \rfloor}{\mathtt{R}} \right\rrbracket q \right| & \leq \frac{q}{2}\left(5 + \frac{|a|}{\mathtt{R}}\right), \\ \left| ab - \left\llbracket \frac{a \lfloor b\mathtt{R}/q \rfloor}{\mathtt{R}} \right\rrbracket_b q \right| & \leq \frac{q}{2}\left(7 + \frac{|a|}{\mathtt{R}}\right), \end{cases}$$

for integer approximations ${}_b\llbracket\rrbracket, \llbracket\rrbracket_b$ defined in Definitions 33 and 34.

9.2.1.3 Plantard Multiplication

For Plantard multiplication, all the variants are already introduced in Section 3.5. As long as the conditions in Definition 26 are met, the result is the same as Montgomery multiplication, and hence the range follows. We need a middle product computing the middle $\log_2 \mathtt{R}$ bits of a product of a $\log_2 \mathtt{R}$-bit number and a $2 \log_2 \mathtt{R}$-bit number.

9.2.1.4 Required Multiplication Instructions for Modular Multiplications

In this section, we briefly summarize the required multiplication instructions for Montgomery, Barrett, and Plantard multiplications.

Montgomery multiplications. For Montgomery multiplications, we need three multiplication instructions in each of the variants. We need one low multiplication and two long multiplications for the accumulative Montgomery multiplication with long multiplications, one low multiplication and two long multiplications for the subtractive Montgomery multiplication with long multiplications, and one low multiplication and two high multiplications for the subtractive Montgomery multiplication with high multiplications. See Table 9.6 for a summary.

Table 9.6: Overview of multiplications used in Montgomery multiplications. Mont. (long acc.) stands for the accumulative Montgomery multiplication with long multiplications (cf. Definition 28), Mont. (long sub.) stands for the subtractive Montgomery multiplication with long multiplications (cf. Definition 29), and Mont. (high sub.) stands for the subtractive Montgomery multiplication with high multiplications (cf. Definition 30).

Instruction	Mont. (long acc.)	Mont. (long sub.)	Mont. (high sub.)
Low	1	1	1
Low (acc.)	0	0	0
Low (sub.)	0	0	0
High	0	0	1
High (acc.)	0	0	0
High (sub.)	0	0	1
Long	1	1	0
Long (acc.)	1	0	0
Long (sub.)	0	1	0
Wide	0	0	0
Wide (acc.)	0	0	0

Barrett and Plantard multiplications. For Barrett multiplications, each variant requires two low multiplications and one high multiplication. The variants are distinguished by the accuracy of the high multiplication. For Plantard multiplications, we need two wide multiplications for the middle products. The wide multiplications can also be replaced with high and low multiplications. See Table 9.7 for a summary.

Table 9.7: Overview of multiplications used in Barrett and Plantard multiplications. There are two options for Plantard multiplication: either implementing it with wide multiplications or low and high multiplications.

Instruction	Barrett	Plantard with wide mul.	Plantard
Low	1	0	0
Low (acc.)	0	0	2
Low (sub.)	1	0	0
High	1	0	1
High (acc.)	0	0	1
High (sub.)	0	0	0
Long	0	0	0
Long (acc.)	0	0	0
Long (sub.)	0	0	0
Wide	0	1	0
Wide (acc.)	0	1	0

9.2.2 Armv7-M and Armv7E-M

9.2.2.1 Montgomery Modular Arithmetic

32-bit Montgomery modular arithmetic with `s{mul, mla}l`. We first illustrate 32-bit modular arithmetic in Armv7-M and Armv7E-M. Since there are signed 32-bit long multiplications `s{mul, mla}l`, we can implement Montgomery multiplication as the sequence `smull`, `mla`, `smlal` (cf. Algorithm 9.1). On Cortex-M4, Algorithm 9.1 takes 3 cycles, and on Cortex-M3, Algorithm 9.1 takes variable-time due to the variable-time `s{mul, mla}l`.

32-bit Montgomery modular arithmetic without `s{mul, mla}l`. An alternative approach avoiding `s{mul, mla}l` is to emulate the long multiplications – we compute the long product of two 32-bit registers by first splitting each into 16-bit elements and issuing four 32-bit low multiplications `mul` and four additions with optional carries. See Algorithm 9.2 for an illustration. Furthermore, if the inputs are have absolute values slightly smaller than 2^{31}, we can save an addition with carry as shown in Algorithm 9.3. See Algorithm 9.4 for the resulting 32-bit Montgomery multiplication.

Algorithm 9.1 32-bit Montgomery multiplication in Armv7-M [GKS20, ACC+20].

Input:

$$\begin{cases} \texttt{a} & = a, \\ \texttt{b} & = b, \\ \texttt{q} & = q, \\ \texttt{qprime} & = -q^{-1} \bmod^{\pm} 2^{32}. \end{cases}$$

Output: $\texttt{hi} = \frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right)q}{2^{32}}$.

```
smull lo, hi, a, b
mul    t, lo, qprime
smlal lo, hi, t, q
```

Algorithm 9.2 Emulation of `smlal` (`smull`) with `mul` in Armv7-M.

Input:

$$\begin{cases} (\texttt{alo}, \texttt{ahi}) & = \mathrm{usplit}_{16}(a), \\ (\texttt{blo}, \texttt{bhi}) & = \mathrm{usplit}_{16}(b), \\ (\texttt{clo}, \texttt{chi}) & = \mathrm{usplit}_{32}(c). \end{cases}$$

Output: $(\texttt{clo}, \texttt{chi}) = \mathrm{usplit}_{32}(ab + c)$.

```
mul   lo, alo, blo
mul   hi, ahi, bhi
mul    t, alo, bhi
adds  lo,  lo,   t, lsl #16
adc   hi,  hi,   t, asr #16
mul    t, ahi, blo
adds  lo,  lo,   t, lsl #16
adc   hi,  hi,   t, asr #16
                          ▷ Stop here and return registers lo and hi for smull.
adds clo, clo,  lo
adc  chi, chi,  hi
```

Algorithm 9.3 Macros `sbsmlal` (`sbsmull`) emulating `smull` and `smlal` with `mul`/`mla` for multiplicands with absolute values smaller than $2^{30} + 2^{16} - 1$ in Armv7-M [GKS20].

Input:

$$\begin{cases} (\texttt{alo}, \texttt{ahi}) & = \text{usplit}_{16}(a), \\ (\texttt{blo}, \texttt{bhi}) & = \text{usplit}_{16}(b), \\ (\texttt{clo}, \texttt{chi}) & = \text{usplit}_{32}(c). \end{cases}$$

Output: $(\texttt{clo}, \texttt{chi}) = \text{usplit}_{32}(ab + c)$.

```
mul   lo, alo, blo
mul   hi, ahi, bhi
mul    t, alo, bhi
mla    t, ahi, blo, t
adds  lo,  lo,   t, lsl #16
adc   hi,  hi,   t, asr #16
                       ▷ Stop here and return registers lo and hi for sbsmull.
adds clo, clo,  lo
adc  chi, chi,  hi
```

Algorithm 9.4 Constant-time 32-bit Montgomery multiplication in Armv7-M [GKS20].

Input:

$$\begin{cases} (\texttt{alo}, \texttt{ahi}) & = \text{usplit}_{16}(a), \\ (\texttt{blo}, \texttt{bhi}) & = \text{usplit}_{16}(b), \\ (\texttt{qlo}, \texttt{qhi}) & = \text{usplit}_{16}(q), \\ \texttt{qprime} & = -q^{-1} \bmod^{\pm} 2^{32}. \end{cases}$$

Output: $\texttt{hi} = \frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}}$.

```
sbsmull lo,  hi, alo, ahi, blo, bhi     ▷ $(\texttt{lo}, \texttt{hi}) = \text{usplit}_{32}(ab)$.
mul     ahi,  lo, qprime                ▷ $\texttt{ah} = abq^{-1} \bmod^{\pm} \texttt{R}$.
ubfx    alo, ahi,  #0, #16
asr     ahi, ahi, #16        ▷ $(\texttt{alo}, \texttt{ahi}) = \text{usplit}_{16}\left(-abq^{-1} \bmod^{\pm} \texttt{R}\right)$.
sbsmlal lo,  hi, alo, ahi, qlo, qhi     ▷ $\texttt{hi} = \frac{ab + \left(-abq^{-1} \bmod^{\pm} \texttt{R}\right) q}{\texttt{R}}$.
```

16-bit Montgomery modular arithmetic. One can implement the 16-bit Montgomery multiplication straightforwardly with `s{mul, mla}{b, t}{b, t}` in the DSP extension of Armv7E-M. On Cortex-M3 implementing Armv7-

M with variable-time long multiplications, we implement 16-bit Montgomery multiplication with `mul`, `mla`, and `sbfx`. See Algorithms 9.6 and 9.5 for illustrations.

Algorithm 9.5 16-bit Montgomery multiplication with `mul`/`mla` in Armv7-M.

Input:

$$\begin{cases} \texttt{a} & = a, \\ \texttt{b} & = b, \\ \texttt{q} & = q, \\ \texttt{qprime} & = -q^{-1} \bmod^{\pm} 2^{32}. \end{cases}$$

Output: $\mathrm{shi}_{16}(\texttt{c}) = \frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{16}\right)q}{2^{16}}$.

```
mul  c, a, b
mul  t, c, qprime
sbfx t, t, #0, #16
mla  c, t, q, c          ▷ $\mathrm{shi}_{16}(\texttt{c}) = \frac{ab+\left(-abq^{-1} \bmod^{\pm} 2^{16}\right)q}{2^{16}}$.
```

Algorithm 9.6 16-bit Montgomery multiplication with DSP instructions in Armv7E-M [ABCG20].

Input:

$$\begin{cases} \mathrm{slo}_{16}(\texttt{a}) & = a, \\ \mathrm{slo}_{16}(\texttt{b}) & = b, \\ \texttt{q} & = q, \\ \texttt{qprime} & = -q^{-1} \bmod^{\pm} 2^{32}. \end{cases}$$

Output: $\mathrm{shi}_{16}(\texttt{c}) = \frac{ab + \left(-abq^{-1} \bmod^{\pm} 2^{16}\right)q}{2^{16}}$.

```
smulbb c, a, b
smulbb t, c, qprime
smlabb c, t, q, c        ▷ $\mathrm{shi}_{16}(\texttt{c}) = \frac{ab+\left(-abq^{-1} \bmod^{\pm} 2^{16}\right)q}{2^{16}}$.
```

9.2.2.2 Barrett Modular Arithmetic

32-bit Barrett modular arithmetic with `smmul{, r}`. For 32-bit Barrett multiplication using `smmul{, r}`, we straightforwardly implement the high multiplication with `smmulr` in the standard one and with `smull`/`smmul` for the floor variant. See Algorithms 9.7 for illustrations. If the inputs are 16-bit elements, we can implement 32-bit Barrett multiplication and reduction efficiently with `smlawb` [AHKS22]. See Algorithm 9.8 for an illustration.

Algorithm 9.7 Standard (floor) 32-bit Barrett multiplication in Armv7E-M.

Input:

$$\begin{cases} \mathtt{a} & = a, \\ \mathtt{b} & = b, \\ \mathtt{bp} & = \left\lfloor \frac{2^{32}b}{q} \right\rceil, \\ \mathtt{q} & = q. \end{cases}$$

Output: $\mathtt{c} = ab - \left\lfloor \frac{a\lfloor 2^{32}b/q \rceil}{2^{32}} \right\rceil q.$

```
mul    c,  a,  b
smmulr hi, a,  bp        ▷ Use smmul or smull for the floor variant.
mls    c,  hi, $q$, c
```

Algorithm 9.8 Standard 32-bit Barrett multiplication (reduction) for 16-bit inputs and 32-bit constant with DSP instructions in Armv7E-M.

Input:

$$\begin{cases} \mathrm{shi}_{16}(\mathtt{a}) & = a, \\ \mathtt{b} & = b, \\ \mathtt{bp} & = \left\lfloor \frac{2^{32}b}{q} \right\rceil. \end{cases}$$

Output: $\mathtt{c} = ab - \left\lfloor \frac{a\lfloor 2^{32}b/q \rceil}{2^{32}} \right\rceil q.$

```
smulbb c, a,  b          ▷ Skip this line for 32-bit Barrett reduction.
smlawb t, a,  bp, $2^{31}$
smlatb c, t,  q,  c
```

32-bit Barrett modular arithmetic without `smmul{, r}`. For 32-bit Barrett modular arithmetic without `smmul{, r}`, we emulate the them with `mul`, `mla`, and `mls`. Algorithm 9.9 emulates `smmulr`. If we skip the last addition with 2^{31} in Algorithm 9.9, we have `smmul`. To further the idea, we skip the multiplication of the lower parts, add 2^{15} to one of the middle parts, and accumulate the upper 16-bits of the middle parts to the final result, completely avoiding the carry computation. We can also push the idea even further – skip the addition with 2^{15}. This results in the approximate variant of the Barrett multiplication with 2-limb arithmetic (cf. Section 3.4). See Algorithm 9.10 for the resulting approximations of the high products and Algorithm 9.11 for the approximate variant of 32-bit Barrett multiplication.

Algorithm 9.9 32-bit high multiplication with the integer approximation $\lfloor \rceil$ in Armv7-M.

Input:

$$\begin{cases} (\texttt{alo}, \texttt{ahi}) = \mathrm{usplit}_{16}(a), \\ (\texttt{blo}, \texttt{bhi}) = \mathrm{usplit}_{16}(b). \end{cases}$$

Output: `chi` $= \left\lfloor \frac{ab}{2^{32}} \right\rceil$.

```
mul  clo, alo, blo
mul  chi, ahi, bhi
mul    t, alo, bhi
adds clo, clo,   t, lsl #16
adc  chi, chi,   t, asr #16
mul    t, ahi, blo
adds clo, clo,   t, lsl #16
adc  chi, chi,   t, asr #16
add  chi, chi, clo, lsr #31
```

Algorithm 9.10 Macro `smmulr_approx` implementing the 32-bit high multiplication with integer approximation $\llbracket\rrbracket_b$ (${}_b\llbracket\rrbracket$) in Armv7-M.

Input:

$$\begin{cases} (\texttt{alo}, \texttt{ahi}) = \mathrm{usplit}_{16}(a), \\ (\texttt{blo}, \texttt{bhi}) = \mathrm{usplit}_{16}(b). \end{cases}$$

Output: `chi` $= \left\llbracket \frac{ab}{2^{32}} \right\rrbracket_b$.

```
mul chi, ahi, bhi
mul   t, alo, bhi                 ▷ Add $2^{15}$ to t if we want ${}_b\llbracket\rrbracket$.
add chi, chi,   t, asr #16
mul   t, ahi, blo
add chi, chi,   t, asr #16
```

Algorithm 9.11 Approximate variant of 32-bit Barrett multiplication in Armv7-M [HKS23].

Input:

$$\begin{cases} \texttt{a} & = a, \\ \texttt{b} & = b, \\ (\texttt{blo}, \texttt{bhi}) & = \mathrm{usplit}_{16}\left(\left\lfloor \frac{2^{32}b}{q} \right\rceil\right). \end{cases}$$

Output: $\texttt{c} = ab - \left\llbracket \frac{a\left\lfloor 2^{32}b/q \right\rceil}{2^{32}} \right\rrbracket_b q.$

```
mul           c,  a,   b                       ▷ $\texttt{c} = ab \bmod^{\pm} \texttt{R}$.
ubfx         t0,  a,  #0, #16
asr           a,  a, #16                       ▷ $(\texttt{t0}, \texttt{a}) = \mathrm{usplit}_{16}(a)$.
smmulr_approx t1, t0,   a, blo, bhi, t2        ▷ $\texttt{t1} = \left\llbracket \frac{a\lfloor b\texttt{R}/q \rceil}{\texttt{R}} \right\rrbracket_b$.
mls           c, t1,  $q$,   c                   ▷ $\texttt{c} = ab - \left\llbracket \frac{a\lfloor b\texttt{R}/q \rceil}{\texttt{R}} \right\rrbracket q$.
```

16-bit Barrett modular arithmetic. As for 16-bit Barrett multiplication, see Algorithms 9.12 and 9.13 for straightforward implementations.

Algorithm 9.12 Standard (floor) 16-bit Barrett multiplication in Armv7-M [HKS23].

Input:

$$\begin{cases} \texttt{a} & = a, \\ \texttt{b} & = b, \\ \texttt{bp} & = \left\lfloor \frac{2^{16}b}{q} \right\rceil, \\ \texttt{q} & = q. \end{cases}$$

Output: $\texttt{c} = ab - \left\lfloor \frac{a\left\lfloor 2^{16}b/q \right\rceil}{2^{16}} \right\rceil q.$

```
mul c, a,  b
mla t, a, bp, #0x8000            ▷ Use mul here for the floor variant.
asr t, t, #16
mla c, t,  q, c
```

Algorithm 9.13 Standard (floor) 16-bit Barrett multiplication with DSP instructions in Armv7E-M.

Input:

$$\begin{cases} \mathrm{slo}_{16}(\mathtt{a}) & = a, \\ \mathrm{usplit}_{16}(\mathtt{b}) & = \left(b, \left\lfloor \frac{2^{16}b}{q} \right\rceil\right). \end{cases}$$

Output: $\mathtt{c} = ab - \left\lfloor \frac{a\lfloor 2^{16}b/q \rceil}{2^{16}} \right\rceil q.$

```
smulbb c, a, b
smlabb t, a, b, #0x8000      ▷ Use smulbb here for the floor variant.
smlatb c, t, q, c
```

9.2.2.3 Plantard Modular Arithmetic

Algorithm 9.14 16-bit Plantard multiplication in Armv7-M (based on [AMOT22, HZZ+24]).

Input:

$$\begin{cases} \mathtt{a} & = a, \\ \mathtt{bp} & = -bq^{-1} \bmod^{\pm} 2^{32}, \\ \mathtt{t} & = \left\lfloor \frac{2^{15}}{q} \right\rfloor, \\ \mathtt{q} & = q. \end{cases}$$

Output: $\mathrm{shi}_{16}\left(\mathtt{c}\right) = \left\lfloor \frac{\left(\left\lfloor -abq^{-1} \bmod^{\pm} 2^{32}/2^{16} \right\rfloor + \left\lfloor 2^{15}/q \right\rfloor\right) q}{2^{16}} \right\rfloor.$

```
mul c, bp, a
add c, t, c, asr #16      ▷ $c = \left\lfloor \frac{2^{15}}{q} \right\rfloor + \left\lfloor \frac{-abq^{-1} \bmod^{\pm} 2^{32}}{2^{16}} \right\rfloor$.
mul c, c, q
```

16-bit Plantard modular arithmetic. Algorithm 9.14 implements the 16-bit Plantard multiplication in Armv7-M. [AMOT22] proposed the 32-bit signed Plantard multiplication with 64-bit multiplication instructions, which clearly transfers to the 16-bit version. [HZZ+24] later implemented the last rounding with barrel shifter while adding with a power of two. Notice that in Algorithm 9.14, the power of two is replaced by a slightly larger non-power-of-two constant. As for the Armv7E-M implementation, we use the instruction `smulw{b, t}` as shown in Algorithm 9.15 proposed by [HZZ+22]. Notice that in [HZZ+22], the last rounding adds the largest power-of-two-multiple of the q

that is smaller than 2^{15} to the register. We replace the power-of-two-multiple by 2^{15} proposed by [AMOT22] for simplicity.

Algorithm 9.15 16-bit Plantard multiplication with DSP instructions in Armv7E-M (based on [AMOT22, HZZ⁺22]).

Input:

$$\begin{cases} \mathrm{slo}_{16}(\mathtt{a}) & = a, \\ \mathtt{bp} & = -bq^{-1} \bmod^{\pm} 2^{32}, \\ \mathtt{q} & = q. \end{cases}$$

Output: $\mathrm{shi}_{16}(\mathtt{c}) = \left\lfloor \frac{\left\lfloor -abq^{-1} \bmod^{\pm} 2^{32}/2^{16} \right\rceil q}{2^{16}} \right\rceil$.

```
smulwb c, bp, a
smlabb c, c, q, #0x8000        ▷ Replace 0x8000 by other constants for
    other rounding versions.
```

32-bit Plantard modular arithmetic. For the 32-bit Plantard multiplication, Algorithm 9.16 implements the proposal by [AMOT22] with the DSP instructions `smmlar` and `smmulr` in Armv7E-M.

Algorithm 9.16 32-bit Plantard multiplication with DSP instructions in Armv7E-M (adapted from [AMOT22]).

Input:

$$\begin{cases} \mathtt{a} & = a, \\ (\mathtt{blo}, \mathtt{bhi}) & = \mathrm{usplit}_{32}\left(-bq^{-1} \bmod^{\pm} 2^{64}\right), \\ \mathtt{q} & = q. \end{cases}$$

Output: $\mathtt{c} = \left\lfloor \frac{\left\lfloor -abq^{-1} \bmod^{\pm} 2^{64}/2^{32} \right\rceil q}{2^{32}} \right\rceil$.

```
mul    c, a, bhi
smmlar c, a, blo, c            ▷ $\mathtt{c} = \left\lfloor \frac{-abq^{-1} \bmod^{\pm} 2^{64}}{2^{32}} \right\rceil$.
          ▷ Use smmulr or smull if we want $\left\lfloor \frac{-abq^{-1} \bmod^{\pm} 2^{64}}{2^{32}} \right\rfloor$.
smmulr c, c,  q                ▷ $\left\lfloor \frac{\left\lfloor -abq^{-1} \bmod^{\pm} 2^{64}/2^{32} \right\rceil q}{2^{32}} \right\rceil$.
          ▷ Use smlal for other rounding versions.
```

9.2.2.4 Timings

Cortex-M3 timings. For the Cortex-M3 timings, since long multiplications take variable-time cycles, 32-bit modular multiplications such as Algorithms 9.1 and 9.7 are variable-time. Other approaches are constant-time. See Table 9.8 for a summary of the timings of 32-bit modular multiplications and Table 9.9 for a summary of the timings of 16-bit modular multiplications.

Table 9.8: Overview of 32-bit modular multiplications with 32-bit input values on Cortex-M3. Cycles are obtained by summing up the instruction timings from the manual [ARM10a].

Operation	Work	Implementation	Cycle
Constant-Time			
Montgomery mul.	[GKS20]	Algorithm 9.4	23
Barrett mul. (approx.)	[HKS24]	Algorithm 9.11	12
Variable-Time			
Montgomery mul.	[GKS20]	Algorithm 9.1	9 − 16
Barrett mul. (floor)	[HKS24]	Algorithm 9.7	6 − 8

Table 9.9: Overview of 16-bit modular multiplications with 16-bit input values on Cortex-M3. Cycles are obtained by summing up the instruction timings from the manual [ARM10a].

Operation	Work	Implementation	Cycle
Montgomery mul.	[GKS20]	Algorithm 9.5	5
Barrett mul. (standard)	[HKS24]	Algorithm 9.12	6
Barrett mul. (floor)	[HKS24]	Algorithm 9.12	5
Plantard mul.	[AMOT22, HZZ$^+$24]	Algorithm 9.14	3

Cortex-M4 timings. For Cortex-M4 timings, since each arithmetic instructions take 1 cycle, the timings straightforwardly follow from the number of instructions. See Table 9.10 for a summary of the timings of 32-bit modular multiplications and Table 9.11 for a summary of the timings of 16-bit modular multiplications.

Table 9.10: Overview of 32-bit modular multiplications with 32-bit input values on Cortex-M4. Cycles are obtained by summing up the instruction timings from the manual [ARM10b, ARM10c].

Operation	Work	Implementation	Cycle
Montgomery mul.	[GKS20, ACC$^+$20]	Algorithm 9.1	3
Barrett reduction	[AHKS22]	Algorithm 9.8	2
Barrett mul.	[AHKS22, HKS24]	Algorithms 9.7 and 9.8	3
Plantard mul.	[AMOT22]	Algorithm 9.16	3

Table 9.11: Overview of 16-bit modular multiplications with packed 16-bit input values on Cortex-M4. Cycles are obtained by summing up the instruction timings from the manual [ARM10b, ARM10c] averaged over each 32-bit registers.

Operation	Work	Implementation	Cycle
Montgomery multiplication	[ABCG20]	Algorithm 9.6	3
Barrett multiplication	[HKS24]	Algorithm 9.13	3
Plantard multiplication	[HZZ$^+$22]	Algorithm 9.15	2

9.2.3 Armv8-A Neon

In Armv8-A Neon, the design of efficient modular multiplications amounts to identifying suitable mapping to high multiplication instructions. This is not straightforward as we only have high multiplications with doubling and optional rounding.

9.2.3.1 Montgomery Modular Arithmetic

For vectorized Montgomery multiplication in Neon, the state-of-the-art approach implements the subtractive version by [Sei18] computing the difference of the high products. In Neon, the only possible high multiplication is `sqdmulh` computing the 2-multiple with saturation of the desired high product. To mitigate the additional 2-scaling, we replace the subtraction with the halving variant `shsub`. See Algorithm 9.17 for an illustration.

Algorithm 9.17 Montgomery multiplication in Armv8-A Neon [SKS⁺21, BHK⁺21].

Inputs:

$$\begin{cases} \mathtt{a} &= (a_i)\,, \\ \mathtt{b} &= (b_i)\,, \\ \mathtt{bp} &= \left(b_i q^{-1} \bmod^{\pm} \mathtt{R}\right), \\ \mathtt{q} &= q. \end{cases}$$

Output: $\mathtt{c} = \left(\frac{1}{2}\left(\left\lfloor \frac{2a_ib_i}{\mathtt{R}} \right\rceil - \left\lfloor \frac{2\left(a_ib_iq^{-1} \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} \right\rceil\right)\right).$

```
1: sqdmulh  c,  a,  b      ▷ $\mathtt{c} = \left(\left\lfloor \frac{2a_ib_i}{\mathtt{R}} \right\rceil\right)$.
2: mul      t,  a,  bp     ▷ $\mathtt{t} = \left(a_ib_iq^{-1} \bmod^{\pm}\mathtt{R}\right)$.
3: sqdmulh  t,  t,  q      ▷ $\mathtt{t} = \left(\left\lfloor \frac{2\left(a_ib_iq^{-1} \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} \right\rceil\right)$.
4: shsub    c,  c,  t      ▷ $\mathtt{c} = \left(\frac{1}{2}\left(\left\lfloor \frac{2a_ib_i}{\mathtt{R}} \right\rceil - \left\lfloor \frac{2\left(a_ib_iq^{-1} \bmod^{\pm}\mathtt{R}\right)q}{\mathtt{R}} \right\rceil\right)\right)$.
```

9.2.3.2 Barrett Modular Arithmetic

Algorithm 9.18 w-bit Barrett reduction for $w \geq \log_2 \mathtt{R}$ in Armv8-A Neon [BHK⁺21].

Inputs:

$$\begin{cases} \mathtt{a} &= (a_i)\,, \\ \mathtt{m} &= \left\lfloor \frac{2^w}{q} \right\rceil, \\ \mathtt{q} &= q. \end{cases}$$

Output: $\mathtt{a} = \left(a_i - \left\lfloor \frac{a_i \lfloor 2^w/q \rceil}{2^w} \right\rceil q\right).$

```
1: sqdmulh t, a, m
2: srshr   t, t, #$(w + 1 - \log_2 \mathtt{R})$
3: mls     a, t, q
```

For Barrett reduction, we can also mitigate the additional 2-scaling with the rounding version of arithmetic right-shift **`srshr`**. See Algorithm 9.18 for an illustration. As for Barrett multiplication, we replace the precomputed constant $\left\lfloor \frac{b_i \texttt{R}}{q} \right\rceil$ by $\frac{\lfloor b_i \texttt{R}/q \rceil_2}{2}$ so the rounding is applied to the correct precision as shown in Algorithm 9.19. Since $\lfloor \rceil_2$ is a 1-integer approximation, the result differs to the standard one by at most q in absolute value.

Algorithm 9.19 Barrett multiplication in Neon [BHK$^+$21, Algorithm 10].

Inputs:

$$\begin{cases} \texttt{a} & = (a_i) \,, \\ \texttt{b} & = (b_i) \,, \\ \texttt{bhi} & = \left(\frac{\lfloor b_i \texttt{R}/q \rceil_2}{2} \right) , \\ \texttt{q} & = q . \end{cases}$$

Output: `lo` $= \left(a_i b_i - \left\lfloor \frac{a_i \lfloor b_i \texttt{R}/q \rceil_2}{\texttt{R}} \right\rceil q \right)$.

```
mul      lo, a,  b       ▷ lo = $(a_i b_i)$.
sqrdmulh hi, a,  bhi     ▷ hi = $\left( \left\lfloor \frac{a_i \lfloor b_i \texttt{R}/q \rceil_2}{\texttt{R}} \right\rceil \right)$.
mls      lo, hi, q       ▷ lo = $\left( a_i b_i - \left\lfloor \frac{a_i \lfloor b_i \texttt{R}/q \rceil_2}{\texttt{R}} \right\rceil q \right)$.
```

9.2.3.3 Timings

Table 9.12: Overview of Montgomery and Barrett reductions/multiplications with Armv8-A Neon. Cycles spent on the ports with the heaviest workload are reported.

	Implementation	Cortex-A72	Firestorm
Montgomery mul.	Algorithm 9.17	6	1
Barrett reduction	Algorithm 9.18	6	0.75
Barrett mul.	Algorithm 9.19	6	0.75

See Table 9.12 for an overview of modular multiplications with Neon and the timings on Cortex-A72 and Firestorm.

9.2.4 AVX2

In AVX2, we only have 16-bit high multiplications `vpmulhw` and `vpmulhrsw`. For 16-bit modular arithmetic, we can similarly compute with 16-bit high multiplications. As for 32-bit modular arithmetic, we have to resort to 32-bit long multiplications and issue several permutations to extract the desired 32-bit elements from the long products.

9.2.4.1 Montgomery Modular Arithmetic

16-bit Montgomery modular arithmetic. Similarly to the Montgomery multiplication in Neon, the state-of-the-art approach is the subtractive variant using `vpmulhw` as shown in Algorithm 9.20.

32-bit Montgomery modular arithmetic. As for the 32-bit Montgomery multiplication, the state-of-the-art approach amounts to several 32-bit long multiplications and permutations. We duplicate the odd-indexed 32-bit elements to all the 32-bit elements in each 64-bit elements with `vmovshdup` and combine the odd-indexed 32-bit elements in each resulting 64-bit elements with `vpblendd $0xaa`. See Algorithm 9.21 for an illustration.

Algorithm 9.20 16-bit Montgomery multiplication in AVX2 [Sei18].

Input:

$$\begin{cases} \mathtt{a} & = (a_i), \\ \mathtt{b} & = (b_i), \\ \mathtt{bp} & = \left(b_i q^{-1} \bmod^{\pm} 2^{16}\right), \\ \mathtt{q} & = q. \end{cases}$$

Output: $\mathtt{hi} = \left(\left\lfloor \frac{a_i b_i}{2^{16}} \right\rfloor - \left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{16}\right) q}{2^{16}} \right\rfloor \right)$.

```
1: vpmulhw  b,  a, hi     ▷ $\mathtt{hi} = \left( \left\lfloor \frac{a_i b_i}{2^{16}} \right\rfloor \right)$.
2: vpmullw bp,  a, lo     ▷ $\mathtt{lo} = \left( a_i b_i q^{-1} \bmod^{\pm} 2^{16} \right)$.
3: vpmulhw  q, lo, lo     ▷ $\mathtt{lo} = \left( \left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{16}\right) q}{2^{16}} \right\rfloor \right)$.
4: vpsubw  lo, hi, hi     ▷ $\mathtt{hi} = \left( \left\lfloor \frac{a_i b_i}{2^{16}} \right\rfloor - \left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{16}\right) q}{2^{16}} \right\rfloor \right)$.
```

Algorithm 9.21 32-bit Montgomery multiplication in AVX2 [ABD$^+$20a].

Input:

$$\begin{cases} \texttt{a} & = (a_i), \\ \texttt{b} & = (b_i), \\ \texttt{bp} & = \left(b_i q^{-1} \bmod^{\pm} 2^{32}\right), \\ \texttt{q} & = q. \end{cases}$$

Output: $\texttt{a} = \left(\left\lfloor \frac{a_i b_i}{2^{32}} \right\rfloor - \left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}} \right\rfloor \right)$.

```
vpmuldq        b,    a, hi0                ▷ hi0 = $(a_{2i}b_{2i})$.
vmovshdup      a,    t
vpmuldq        b,    t, hi1                ▷ hi1 = $(a_{2i+1}b_{2i+1})$.
vmovshdup    hi0,  hi0
vpblendd   $0xaa,  hi1, hi0, hi0           ▷ hi0 = $\left(\left\lfloor \frac{a_i b_i}{2^{32}} \right\rfloor\right)$.
vpmuldq       bp,    a,   a                ▷ a = $\left(a_{2i}\left(b_{2i}q^{-1} \bmod^{\pm} 2^{32}\right)\right)$.
vpmuldq       bp,    t,   t                ▷ t = $\left(a_{2i+1}\left(b_{2i+1}q^{-1} \bmod^{\pm} 2^{32}\right)\right)$.
vpmuldq        q,    a,   a                ▷ a = $\left(\left(a_{2i}b_{2i}q^{-1} \bmod^{\pm} 2^{32}\right) q\right)$.
vpmuldq        q,    t,   t                ▷ t = $\left(\left(a_{2i+1}b_{2i+1}q^{-1} \bmod^{\pm} 2^{32}\right) q\right)$.
vmovshdup      a,    a
vpblendd   $0xaa,    t,   a, a             ▷ a = $\left(\left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}} \right\rfloor\right)$.
vpsubd         a,  hi0,   a                ▷ a = $\left(\left\lfloor \frac{a_i b_i}{2^{32}} \right\rfloor - \left\lfloor \frac{\left(a_i b_i q^{-1} \bmod^{\pm} 2^{32}\right) q}{2^{32}} \right\rfloor\right)$.
```

9.2.4.2 Barrett Modular Arithmetic

In the standard Barrett reduction and multiplication, we need to round the high product. This can be achieved with `vpmulhrsw` – for a 16-bit integer a and a positive integer $w < 16$, we rewrite $\left\lfloor \frac{a}{2^w} \right\rceil = \left\lfloor \frac{2^{15-w} a}{2^{15}} \right\rceil$ and implement with `vpmulhrsw` 2^{15-w}. Similarly, when $17 \leq w < 32$, $\left\lfloor \frac{a}{2^w} \right\rceil$ can be implemented with `vpmulhw` and `vpmulhrsw` 2^{31-w}. See Algorithm 9.22 for the resulting standard and floor variant of w-bit Barrett reductions. As for Barrett multiplication, we implement standard 15-bit and the floor variant of 16-bit Barrett multiplications as shown in Algorithm 9.23.

Algorithm 9.22 Standard w-bit Barrett reduction for $17 \leq w < 32$ (floor variant for $16 \leq w < 32$) in AVX2 [Sei18].

Input:

$$\begin{cases} \texttt{a} & = (a_i), \\ \texttt{m} & = \left\lfloor \frac{2^w}{q} \right\rceil, \\ \texttt{q} & = q. \end{cases}$$

Output: $\texttt{a} = \left(a_i - \left\lfloor \frac{a_i \lfloor 2^w/q \rfloor}{2^w} \right\rceil q \right)$.

```
vpmulhw    m, a, t                 ▷ $\texttt{t} = \left( \left\lfloor \frac{a_i \lfloor 2^w/q \rfloor}{2^{16}} \right\rfloor \right)$.
vpmulhrsw  $2^{31-w}$, t, t        ▷ Use vpsraw #$(w-16)$ for the floor variant.
vpmullw    q, t, t
vpsubw     t, a, a
```

Algorithm 9.23 Standard 15-bit (floor in the case of 16-bit) Barrett multiplication in AVX2 (based on [BHK+21]).

Input:

$$\begin{cases} \texttt{a} & = (a_i), \\ \texttt{b} & = (b_i), \\ \texttt{bp} & = \left(\left\lfloor \frac{2^{15} b_i}{q} \right\rceil \right), \\ \texttt{q} & = q. \end{cases}$$

Output: $\texttt{lo} = \left(a_i b_i - \left\lfloor \frac{a_i \lfloor 2^{15} b_i/q \rfloor}{2^{15}} \right\rceil q \right)$.

```
vpmullw    b,  a,  lo
vpmulhrsw  bp, a,  hi    ▷ Use vpmulhw $\left( \left\lfloor \frac{2^{16} b_i}{q} \right\rceil \right)$ for the floor variant.
vpmullw    q,  hi, hi
vpsubw     hi, lo, lo
```

9.2.4.3 Timings

See Table 9.13 for an overview of modular multiplications with AVX2 and the timings on Haswell.

Table 9.13: Overview of Montgomery and Barrett reductions/multiplications with AVX2. Cycles spent on the ports with the heaviest workload in the Haswell architecture are reported.

	Work	Implementation	Cycle
16-bit modular arithmetic			
Montgomery multiplication	[Sei18]	Algorithm 9.20	3
Barrett multiplication	This thesis	Algorithm 9.23	3
32-bit modular arithmetic			
Montgomery multiplication	[ABD+20a]	Algorithm 9.21	6
Barrett multiplication	[Sei18]	Algorithm 9.22	3

9.3 Quotients

This section illustrates efficient implementations of the floor and round of quotients. We recall Theorem 2 as follows. Let q and $\mathtt{R}'$ be coprime integers, $[\![]\!]_0, [\![]\!]_1$ be integer approximations with $[\![]\!]_0, [\![]\!]_1 = \lfloor\rfloor, \lfloor\rceil$. For integers a, b, we have

$$\left[\!\!\left[\frac{ab}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}' = q^{-1}\left(ab - ab \bmod^{[\![]\!]_0} q\right) \bmod^{[\![]\!]_1} \mathtt{R}'.$$

This section investigates the special case $b = \mathtt{R}'$, $q \perp \mathtt{R}'$.

Lemma 4. Let q and $\mathtt{R}'$ be coprime integers, $[\![]\!]_0, [\![]\!]_1$ be integer approximations with $[\![]\!]_0, [\![]\!]_1 = \lfloor\rfloor, \lfloor\rceil$, and a, b, l be integers. We have

$$\left[\!\!\left[\frac{(a+lq)\,\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}' = \left[\!\!\left[\frac{a\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}'.$$

Proof.

$$\left[\!\!\left[\frac{(a+lq)\,\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}' = \left(\left[\!\!\left[\frac{a\mathtt{R}'}{q}\right]\!\!\right]_0 + l\mathtt{R}'\right) \bmod^{[\![]\!]_1} \mathtt{R}' = \left[\!\!\left[\frac{a\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}'.$$

□

Since $\left[\!\!\left[\frac{(a+lq)\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}' = \left[\!\!\left[\frac{a\mathtt{R}'}{q}\right]\!\!\right]_0 \bmod^{[\![]\!]_1} \mathtt{R}'$ for an arbitrary integer l, any representatives of the same equivalence class in $\mathbb{Z}_q$ are mapped to the same value, and we can start with any representatives. Now suppose $[\![\,]\!]_0 = \lfloor\,\rceil$. We identify a sufficiently large power of two $\mathtt{R}$ implementing

$$\left\lfloor \frac{a\mathtt{R}'}{q} \right\rceil = \left\lfloor \frac{a\left\lfloor \mathtt{R}'\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil$$

for a a representative of an equivalence class in $\mathbb{Z}_q$. There are two cases: $a \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ and $a \in [0, q) \cap \mathbb{Z}$. We first prove the following.

Lemma 5. For a real number r, if $r \in \mathbb{Q} - \left(\frac{1}{2} + \mathbb{Z}\right)$, then $\lfloor r \rceil = -\lfloor -r \rceil$.

Proof. We first observe that $\lfloor r + 0.5 \rfloor = \lceil r - 0.5 \rceil$. This implies

$$\lfloor r \rceil = \lfloor r + 0.5 \rfloor = \lceil r - 0.5 \rceil = -\lfloor -r + 0.5 \rfloor = -\lfloor r \rceil .$$

□

Theorem 11. Let q be an odd integer, $\mathtt{R} \perp q$, $\mathtt{R}' \perp q$ be integers. For an integer a, we have

$$\left(\left\lfloor \frac{a\mathtt{R}'}{q} \right\rceil = \left\lfloor \frac{a\left\lfloor \mathtt{R}'\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil \right) \longrightarrow \left(\left\lfloor \frac{-a\mathtt{R}'}{q} \right\rceil = -\left\lfloor \frac{a\left\lfloor \mathtt{R}'\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil \right).$$

Proof.

$$\left\lfloor \frac{-a\mathtt{R}'}{q} \right\rceil = -\left\lfloor \frac{a\mathtt{R}'}{q} \right\rceil = -\left\lfloor \frac{a\left\lfloor \mathtt{R}'\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil .$$

□

By Theorem 11, the correctness of a computation with negative inputs follow from the positive ones. This implies a much more aggressive choice of $\mathtt{R}$ if $a \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ while implementing $\left\lfloor \frac{a\mathtt{R}'}{q} \right\rceil = \left\lfloor \frac{a\left\lfloor \mathtt{R}'\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil$.

9.3.1 Compressions in Kyber

Table 9.14: Smallest $\mathtt{R}$s implementing $\left\lfloor \frac{a2^d}{q} \right\rceil = \left\lfloor \frac{a\left\lfloor 2^d\mathtt{R}/q \right\rfloor}{\mathtt{R}} \right\rceil$ for various d's.

2^d	$a \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$		$a \in [0, q) \cap \mathbb{Z}$	
	$\mathtt{R}$	Computation	$\mathtt{R}$	Computation
2	2^{19}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{19}} \right\rceil$	2^{19}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{19}} \right\rceil$
4	2^{18}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{18}} \right\rceil$	2^{18}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{18}} \right\rceil$
8	2^{17}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{17}} \right\rceil$	2^{17}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{17}} \right\rceil$
16	2^{16}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{16}} \right\rceil$	2^{16}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{16}} \right\rceil$
32	2^{15}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{15}} \right\rceil$	2^{15}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{15}} \right\rceil$
64	2^{14}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{14}} \right\rceil$	2^{14}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{14}} \right\rceil$
128	2^{13}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{13}} \right\rceil$	2^{13}	$\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rfloor}{2^{13}} \right\rceil$
256	2^{21}	$\left\lfloor \frac{a\left\lfloor 2^{29}/q \right\rfloor}{2^{21}} \right\rceil$	2^{21}	$\left\lfloor \frac{a\left\lfloor 2^{29}/q \right\rfloor}{2^{21}} \right\rceil$
512	2^{20}	$\left\lfloor \frac{a\left\lfloor 2^{29}/q \right\rfloor}{2^{20}} \right\rceil$	2^{23}	$\left\lfloor \frac{a\left\lfloor 2^{32}/q \right\rfloor}{2^{23}} \right\rceil^*$
1024	2^{22}	$\left\lfloor \frac{a\left\lfloor 2^{32}/q \right\rfloor}{2^{22}} \right\rceil^*$	2^{23}	$\left\lfloor \frac{a\left\lfloor 2^{33}/q \right\rfloor}{2^{23}} \right\rceil^*$
2048	2^{21}	$\left\lfloor \frac{a\left\lfloor 2^{32}/q \right\rfloor}{2^{21}} \right\rceil^*$	2^{22}	$\left\lfloor \frac{a\left\lfloor 2^{33}/q \right\rfloor}{2^{22}} \right\rceil^*$

* Overflow with 32-bit arithmetic.

Barrett-based compression. We apply the ideas to the compressions in Kyber. For a positive integer d and $q = 3329$, we define

$$\mathtt{Compress}_d : \begin{cases} [0, q) \cap \mathbb{Z} & \rightarrow \quad [0, 2^d) \cap \mathbb{Z}, \\ a & \mapsto \quad \left\lfloor \frac{a2^d}{q} \right\rceil \bmod^{+} 2^d. \end{cases}$$

In Kyber, we need to implement $\mathtt{Compress}_d$ for $d = 1, 4, 5, 10, 11$. As explained in Lemma 4, if we replace the domain of $\mathtt{Compress}_d$ by $\left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$, the results are the same for inputs differed by a multiple of q. Following Corollary 5, if $|a| < \frac{\mathtt{R}}{2\left|b\mathtt{R} \bmod^{\pm} q\right|}$, we have $\left\lfloor \frac{ab}{q} \right\rceil = \left\lfloor \frac{a\lfloor b\mathtt{R}/q \rceil}{\mathtt{R}} \right\rceil$, implying $\mathtt{R} = 2^{23}$ suffices for $a \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ and $\mathtt{R} = 2^{24}$ suffices for $a \in [0, q) \cap \mathbb{Z}$. We can also brute-force all the inputs and find the smallest $\mathtt{R}$s implementing $\left\lfloor \frac{a2^d}{q} \right\rceil = \left\lfloor \frac{a\left\lfloor 2^d\mathtt{R}/q \right\rceil}{\mathtt{R}} \right\rceil$ for $d = 1, \ldots, 11$ as shown in Table 9.14. Since the resulting computation is used in Barrett multiplication, we call the computation the Barrett-based compression.

$\mathtt{Compress}_{\{1,4,5\}}$. As shown in Table 9.14, we can implement $\mathtt{Compress}_d$ as $\left\lfloor \frac{a\left\lfloor 2^{20}/q \right\rceil}{2^{20-d}} \right\rceil \bmod {}^{+}2^d$ for $d = 1, 4, 5$. See Listing 9.1 for illustrations.

Listing 9.1: C implementations of $\mathtt{Compress}_{\{1,4,5\}}$ for $\mathtt{a} \in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ where $315 = \left\lfloor \frac{2^{20}}{q} \right\rceil$, and $q = 3329$.

```c
int16_t compress1(const int16_t a){
    return (((int32_t)a * 315 + (1 << 18)) >> 19) & 0x1;
}

int16_t compress4(const int16_t a){
    return (((int32_t)a * 315 + (1 << 15)) >> 16) & 0xf;
}

int16_t compress5(const int16_t a){
    return (((int32_t)a * 315 + (1 << 14)) >> 15) & 0x1f;
}
```

$\mathtt{Compress}_{\{10,11\}}$. As for $\mathtt{Compress}_{10}$ and $\mathtt{Comrpess}_{11}$, we need to shift the product $a\left\lfloor \frac{2^{32}}{q} \right\rceil$ by at least 1 bit prior to rounding since the additions in the rounding will result in overflows in the numerator. See Listing 9.2 for illustrations.

Listing 9.2: C implementations of `Compress`$_{\{10,11\}}$ for `a` $\in \left[-\frac{q}{2}, \frac{q}{2}\right) \cap \mathbb{Z}$ where $1290167 = \left\lfloor \frac{2^{32}}{q} \right\rceil$, and $q = 3329$.

```
int16_t compress10(const int16_t a){
    return ( ((((int32_t)a * 1290167) >> 1) + \
                    (1 << 20)) >> 21) & 0x3ff;
}

int16_t compress11(const int16_t a){
    return ( ((((int32_t)a * 1290167) >> 1) + \
                    (1 << 19)) >> 20) & 0x7ff;
}
```

9.3.2 Armv7-M and Armv7E-M

`Compress`$_d$ with Armv7-M. For `Compress`$_d$ with $d = 1, \ldots, 9$, we straightforwardly translate the C implementations into assembly. Algorithm 9.24 demonstrates the cases $d = 1, \ldots, 7$ and the cases $d = 8, 9$ can be implemented in the same way with different constants. As for the cases $d = 10, 11$, we implement with Barrel shifter as shown in Algorithm 9.25.

Algorithm 9.24 Armv7-M implementation of `Compress`$_d$ for $d = 1, \ldots, 7$.

```
Input: a = $a$.
Output: a = $\left\lfloor \frac{a \lfloor 2^{20}/q \rceil}{2^{20-d}} \right\rceil \bmod^+ 2^d$.
mla a, a, $\left\lfloor \frac{2^{20}}{q} \right\rceil$, $2^{19-d}$
ubfx a, a, #$20 - d$, #$d$
```

Algorithm 9.25 Armv7-M implementation of `Compress`$_d$ for $d = 10, 11$.

```
Input: a = $a$.
Output: a = $\left\lfloor \frac{a \lfloor 2^{32}/q \rceil}{2^{32-d}} \right\rceil \bmod^+ 2^d$.
mul a, a, $\left\lfloor \frac{2^{32}}{q} \right\rceil$
add a, $2^{30-d}$, a, asr #1
ubfx a, a, #$31 - d$, #$d$
```

`Compress`$_d$ with `smmulr`. If we choose `R` $= 2^{32}$, then `smmulr` implements `Compress`$_d$ whenever $\left\lfloor \frac{2^d \mathtt{R}}{q} \right\rceil$ can be stored as a signed 32-bit word. This is the case for $d = 1, \dots, 10$. As for $d = 11$, $31 < \left\lfloor \frac{2^{11} \mathtt{R}}{q} \right\rceil$ so we must choose an `R` $< 2^{32}$. Finally, we implement $\bmod^+ 2^d$ with `ubfx`. See Algorithm 9.26 for an illustration of the resulting implementation.

Algorithm 9.26 Armv7E-M implementation of `Compress`$_d$ with `smmulr` for $d = 1, \dots, 10$.

Input: `a` $= a$.

Output: `a` $= \left\lfloor \frac{a \left\lfloor 2^{32+d}/q \right\rceil}{2^{32}} \right\rceil \bmod^+ 2^d$.

```
1: smmulr a, a, $\left\lfloor \frac{2^{32+d}}{q} \right\rceil$
2: ubfx   a, a, #0, #$d$
```

`Compress`$_d$ with `smlaw{b, t}`. In general, `smlaw{b, t}` implements $a \mapsto \left\lfloor \frac{a \left\lfloor 2^d/q \right\rceil}{\mathtt{R}} \right\rceil$ whenever $\left\lfloor \frac{2^d \mathtt{R}}{q} \right\rceil$ can be stored as a signed 32-bit word. Therefore, we can flexibly choose an `R` for each d. See Algorithm 9.27 for an illustration. Another benefit of `smlaw{b, t}` is the saving of load operations – instead of loading halfwords with `ldrsh`, we load a word with `ldr` and specify the desired halfwords with `smlawb` and `smlawt`.

Algorithm 9.27 Armv7E-M implementation of `Compress`$_d$ with `smlaw{b, t}` for $d = 1, \dots, 11$.

Input: $a = \mathrm{lo}_{16}(\mathtt{a})$.

Output: `t` $= \left\lfloor \frac{a \left\lfloor 2^{32+d}/q \right\rceil}{2^{32}} \right\rceil \bmod^+ 2^d$.

```
1: smlawb t, $\left\lfloor \frac{2^{32+d}}{q} \right\rceil$, a, $2^{15}$     ▷ Use smlawt if the input is $\mathrm{hi}_{16}(\mathtt{a})$.
2: ubfx   t, t, #0, #$d$
```

Cortex-M3 and Cortex-M4 timings. On Cortex-M3, we deploy Algorithms 9.24 and 9.25, amounting to 3 cycles in both implementations. On Cortex-M4, although Algorithms 9.26 and 9.27 amount to same cycles, we prefer Algorithm 9.27 over Algorithm 9.26 due to the saving on load operations – While implementing with `smlaw{b, t}`, we load two 16-bit halfwords to a 32-bit register and multiply the desired ones by alternating between `smlawb` and `smlawt`. On the contrary, we have to perform a load operation for each 16-bit halfwords while implementing with `smmulr`. See Table 9.15 for a summary of cycles.

Table 9.15: Overview of a single $\texttt{Compress}_d$ for $d = 1, 4, 5, 10, 11$ on Cortex-M3 and Cortex-M4. Cycles are obtained by summing up the instruction timings from the manuals [ARM10a, ARM10b, ARM10c].

Operation	Implementation	Cycle
Cortex-M3		
$\texttt{Compress}_1$	Algorithm 9.24	3
$\texttt{Compress}_4$	Algorithm 9.24	3
$\texttt{Compress}_5$	Algorithm 9.24	3
$\texttt{Compress}_{10}$	Algorithm 9.25	3
$\texttt{Compress}_{11}$	Algorithm 9.25	3
Cortex-M4		
$\texttt{Compress}_1$	Algorithm 9.26	2
$\texttt{Compress}_4$	Algorithm 9.26	2
$\texttt{Compress}_5$	Algorithm 9.26	2
$\texttt{Compress}_{10}$	Algorithm 9.26	2
$\texttt{Compress}_{11}$	Algorithm 9.26	2

9.3.3 Armv8-A Neon

In Armv8-A Neon, we only have `sqdmulh` and `sqrdmulh` for high multiplications. We illustrate below vectorized implementations of $\texttt{Compress}_d$ with `sqdmulh` and `sqrdmulh`.

$\texttt{Compress}_{\{1,4,5\}}$ with Armv8-A Neon. For $\texttt{Compress}_d$ with $d = 1, 4, 5$, we choose $\texttt{R} = 2^{20-d}$ so $\left\lfloor \frac{2^d \texttt{R}}{q} \right\rceil = \left\lfloor \frac{2^{20}}{q} \right\rceil = 315$. Since $315 < 2^{15}$, we implement $a \mapsto \left\lfloor \frac{a \left\lfloor 2^d \texttt{R}/q \right\rfloor}{\texttt{R}} \right\rceil$ with `sqdmulh`, `sqrdmulh`, `srshr`, and `shadd`. For $d = 1$, we choose $\texttt{R} = 2^{19}$ and implement $a \mapsto \left\lfloor \frac{a \left\lfloor 2^{20}/q \right\rfloor}{2^{19}} \right\rceil$ with `sqdmulh` and `srshr`. For $d = 4$, we choose $\texttt{R} = 2^{16}$ and implement with `sqdmulh` and rounded 1-bit right-shift. There are two approaches for the rounded 1-bit right-shift: we call `srshr` as in the case $d = 1$ or `shadd`. The latter is more preferable since `shadd` is dispatched to more execution execution ports than `srshr` on some platforms. As for $d = 5$, we choose $\texttt{R} = 2^{15}$ implement with `sqrdmulh`. See Algorithms 9.28, 9.29, and 9.30 for the resulting $\texttt{Compress}_{\{1,4,5\}}$.

Algorithm 9.28 Armv8-A Neon implementation of $\texttt{Compress}_1$.

Input: $\texttt{a} = (a_i)$.

Output: $\texttt{a} = \left(\left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{19}} \right\rceil \bmod^+ 2 \right)$.

```
sqdmulh.8H  a, a, #$\left\lfloor \frac{2^{20}}{q} \right\rceil$
srshr.8H    a, a, #4
and.16B     a, a, #0x1
```

Algorithm 9.29 Armv8-A Neon implementation of $\texttt{Compress}_4$.

Input: $\texttt{a} = (a_i)$.

Output: $\texttt{a} = \left(\left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{16}} \right\rceil \bmod^+ 2^4 \right)$.

```
sqdmulh.8H  a, a, #$\left\lfloor \frac{2^{20}}{q} \right\rceil$
shadd.8H    a, a, #1
and.16B     a, a, #0xf
```

Algorithm 9.30 Armv8-A Neon implementation of $\texttt{Compress}_5$.

Input: $\texttt{a} = (a_i)$.

Output: $\texttt{a} = \left(\left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{15}} \right\rceil \bmod^+ 2^5 \right)$.

```
sqrdmulh.8H a, a, #$\left\lfloor \frac{2^{20}}{q} \right\rceil$
and.16B     a, a, #0x1f
```

$\texttt{Compress}_{\{10,11\}}$ with Armv8-A Neon. For $\texttt{Compress}_d$ with $d = 10, 11$, we choose $\texttt{R} = 2^{32-d}$. This implies $\left\lfloor \frac{2^d \texttt{R}}{q} \right\rceil = \left\lfloor \frac{2^{32}}{q} \right\rceil = 1290167$ is a constant that does not fit into a signed 16-bit halfword and multiplication by $\left\lfloor \frac{2^{32}}{q} \right\rceil$ is split into several instructions. Concretely, we rewrite $\left\lfloor \frac{a\lfloor 2^d\mathtt{R}/q\rfloor}{\mathtt{R}}\right\rceil$ as follows:

$$\begin{aligned}
\left\lfloor \frac{a\left\lfloor 2^d\mathtt{R}/q\right\rfloor}{\mathtt{R}}\right\rceil &= \left\lfloor \frac{\left\lfloor a\left\lfloor 2^d\mathtt{R}/q\right\rfloor/2^{16}\right\rfloor}{2^{16-d}}\right\rceil \\
&= \left\lfloor \frac{\left\lfloor \mathrm{slo}_{16}\left(2^{32}/q\right)a/2^{16}\right\rfloor + \mathrm{shi}_{16}\left(2^{32}/q\right)a}{2^{16-d}}\right\rceil,
\end{aligned}$$

and implement $\left\lfloor \frac{\mathrm{slo}\left(2^{32}/q\right)a}{2^{16}}\right\rfloor$ with **`sqdmulh`** and **`shadd`**. Since the input is a signed 16-bit halfword, we must issue a signed multiplication for the lower part and split the constant $\left\lfloor \frac{2^{32}}{q}\right\rceil$ into signed lower and upper halfwords as $\mathrm{slo}_{16}\left(\left\lfloor \frac{2^{32}}{q}\right\rceil\right) = -20553$ and $\mathrm{shi}_{16}\left(\left\lfloor \frac{2^{32}}{q}\right\rceil\right) = 20$. See Algorithm 9.31 for an illustration.

Algorithm 9.31 Armv8-A Neon implementation of `Compress`$_{\{10,11\}}$.

Input: $\mathtt{a} = (a_i)$.

Output: $\mathtt{a} = \left(\left\lfloor \frac{a_i\left\lfloor 2^{32}/q\right\rfloor}{2^{32-d}}\right\rceil \bmod^+ 2^d\right)$.

```
sqdmulh.8H  t, a, #-20553
shadd.8H    t, t, #0
mla.8H      t, a, #20
srshr.8H    t, t, #$(16-d)$
and.16B     a, t, #$(2^d-1)$
```

Table 9.16: Overview of `Compress`$_d$ for $d = 1, 4, 5, 10, 11$ with Armv8-A Neon. Cycles spent on the execution ports with the heaviest workload are reported.

Operation	Implementation	Cortex-A72	Firestorm
`Compress`$_1$	Algorithm 9.28	2	0.75
`Compress`$_4$	Algorithm 9.29	2	0.75
`Compress`$_5$	Algorithm 9.30	2	0.50
`Compress`$_{\{10,11\}}$	Algorithm 9.31	4	1.25

9.3.4 AVX2

For AVX2 implementations of $\texttt{Compress}_d$, the ideas are similar to the Neon implementations. We outline the following differences: (i) We do not have rounded right-shift instruction in AVX2, and implement it with high multiplication with rounding `vpmulhrsw`. (ii) `vpmulhrsw` computes $(a_i, b_i) \mapsto \left\lfloor \frac{a_i b_i}{2^{15}} \right\rceil$. Our choices of `R`s remain the same as in Neon except for $\texttt{Compress}_4$: we choose $\texttt{R} = 2^{16}$ so `vpmulhrsw` can be used for rounding. See Algorithms 9.32, 9.33, 9.34 and 9.35 for illustrations and Table 9.17 for a summary of the timings on Haswell.

Algorithm 9.32 AVX2 implementation of $\texttt{Compress}_1$.

```
Input: $\texttt{a} = (a_i)$.
Output: $\texttt{a} = \left( \left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{19}} \right\rceil \bmod^+ 2 \right)$.
1: vpmulhw    a, a, $\left\lfloor \frac{2^{20}}{q} \right\rceil$        ▷ $\left( \left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{16}} \right\rfloor \right)$.
2: vpmulhrsw  a, a, $2^{12}$        ▷ $\left( \left\lfloor \frac{a_i \lfloor 2^{20}/q \rceil}{2^{19}} \right\rceil \right)$.
3: vpand      a, a, 1        ▷ The last operand consists of 16 copies of 1.
```

Algorithm 9.33 AVX2 implementation of $\texttt{Compress}_4$.

```
Input: $\texttt{a} = (a_i)$.
Output: $\texttt{a} = \left( \left\lfloor \frac{a_i \lfloor 2^{21}/q \rceil}{2^{17}} \right\rceil \bmod^+ 2^4 \right)$.
1: vpmulhw    a, a, $\left\lfloor \frac{2^{21}}{q} \right\rceil$        ▷ $\left( \left\lfloor \frac{a_i \lfloor 2^{21}/q \rceil}{2^{16}} \right\rfloor \right)$.
2: vpmulhrsw  a, a, $2^{14}$        ▷ $\left( \left\lfloor \frac{a_i \lfloor 2^{21}/q \rceil}{2^{17}} \right\rceil \right)$.
3: vpand      a, a, 15        ▷ The last operand consists of 16 copies of 15.
```

Algorithm 9.34 AVX2 implementation of $\texttt{Compress}_5$.

```
Input: $\mathtt{a} = (a_i)$ .
Output: $\mathtt{a} = \left( \left\lceil \frac{a_i \lfloor 2^{20}/q \rfloor}{2^{15}} \right\rfloor \bmod^+ 2^5 \right)$ .
vpmulhrsw a, a, $\left\lfloor \frac{2^{20}}{q} \right\rceil$    ▷ $\left( \left\lceil \frac{a_i \lfloor 2^{20}/q \rfloor}{2^{15}} \right\rfloor \right)$ .
vpand     a, a, 31    ▷ The last operand consists of 16 copies of 31.
```

Algorithm 9.35 AVX2 implementation of $\texttt{Compress}_d$ for $d = 10, 11$.

```
Input: $\mathtt{a} = (a_i)$ .
Output: $\mathtt{a} = \left( \left\lceil \frac{a_i \lfloor 2^{32}/q \rfloor}{2^{32-d}} \right\rfloor \bmod^+ 2^d \right)$ .
vpmulhw   t, a, $\mathtt{slo}_{16}\left(\frac{2^{32}}{q}\right)$
vpmullw   a, a, $\mathtt{shi}_{16}\left(\frac{2^{32}}{q}\right)$
vpaddw    a, a, t    ▷ $\mathtt{a} = \left( \left\lfloor \frac{a_i \lfloor 2^{32}/q \rfloor}{2^{16}} \right\rfloor \right)$ .
vpmulhrsw a, a, $2^{d-1}$    ▷ $\mathtt{a} = \left( \left\lceil \frac{a_i \lfloor 2^{32}/q \rfloor}{2^{32-d}} \right\rfloor \right)$ .
vpand     a, a, $2^d - 1$    ▷ The last operand consists of 16 copies of $2^d - 1$.
```

Table 9.17: Overview of $\texttt{Compress}_d$ for $d = 1, 4, 5, 10, 11$ with AVX2. Cycles spent on the execution ports with the heaviest workload in the Haswell architecture are reported.

Operation	Implementation	Cycle
$\texttt{Compress}_1$	Algorithm 9.32	2
$\texttt{Compress}_4$	Algorithm 9.33	2
$\texttt{Compress}_5$	Algorithm 9.34	1
$\texttt{Compress}_{\{10,11\}}$	Algorithm 9.35	3

Chapter 10

General Guide for Optimizing Transformations

10.1 General Optimization Strategies

Before going through the implementations of various transformations, we review some general optimization strategies.

10.1.1 The Support of Vector Arithmetic

In most of the computing devices, we have a series of scalar arithmetic computing one element at a time. An attractive feature of processors on phones and personal computers is the support of vector arithmetic (cf. Sections 8.1.2.3 and 8.1.3.3). In this thesis, the most relevant ones are multiplication instructions reviewed in Tables 9.2, 9.3, and 9.4.

10.1.2 Register Pressure

Once the instructions for the coefficient ring arithmetic is determined, we look into the register pressure while issuing the instructions in batches. This determines how many independent computations can be issued at the same time, which greatly impacts the optimization strategies layer-merging and instruction scheduling in Sections 10.1.3 and 10.1.4.

Table 10.1: Registers for scalar arithmetic.

ISA/Extension	# GPR	GPR bit-size	Elem. bit-size
Armv7-M	16	32	$8, 16, 32$
Armv7-A	16	32	$8, 16, 32$
Armv8-A	32	64	$8, 16, 32, 64$
x86-64	16	64	$8, 16, 32, 64$

Table 10.2: Registers for vector arithmetic. For Armv7E-M, the "SIMD reg." column refers to the general-purpose registers, but the general-purpose registers are effectively treated as SIMD registers in several DSP instructions.

ISA/Extension	# SIMD reg.	SIMD reg. bit-size	Elem. bit-size
Armv7E-M	16	32	$8, 16$
Armv7-A Neon	16	128	$8, 16, 32, 64$
Armv8-A Neon	32	128	$8, 16, 32, 64$
AVX2	16	256	$16, 32$
AVX-512	32	512	$16, 32, 52, 64$

10.1.3 Layer-Merging

Layer-merging is a standard technique for reducing the memory operations. While applying a composition of homomorphisms f_i's, a straightforward way is to implement each f_i's separately. For an f_i, we load elements from memory, apply f_i, and store the results to memory. Layer-merging looks into compositions of a limited number of homomorphisms, and compute the the results depended on a subset of input elements with a single load-store pair for each input-output pair. Suppose we want to apply $f_0 : \mathcal{A}_0 \to \mathcal{A}_1$ followed by $f_1 : \mathcal{A}_1 \to \mathcal{A}_2$. For a target subset $\{r_j\} \subset (f_1 \circ f_0)(\mathcal{A}_0)$, we load the elements $\{a_i\} := (f_1 \circ f_0)^{-1}(\{r_j\})$ from memory, compute $\{r_j\} = (f_1 \circ f_0)(\{a_i\})$, and store $\{r_j\}$ to memory. Compared to applying f_0 and f_1 separately, applying $f_1 \circ f_0$ with layer-merging saves the memory operations storing and loading $f_0(\{a_i\})$ to and from memory. The overall layer-merging strategy heavily relies on the internal structure of each homomorphisms and the register pressure of the target platforms.

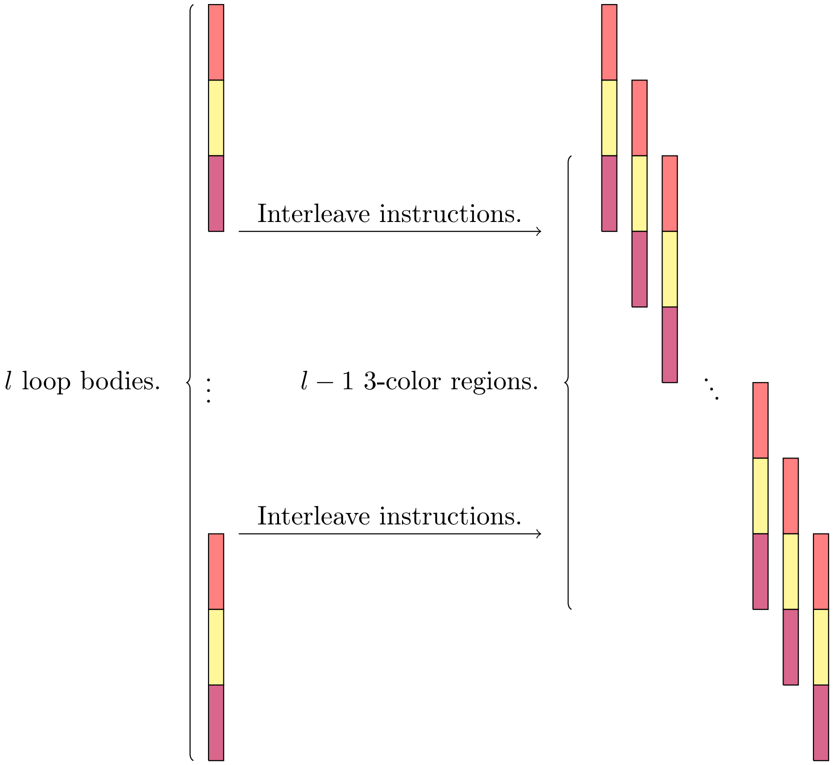
10.1.4 Instruction Scheduling

Categorizing instructions with execution ports. Finally, the last step is to schedule the instructions. For single-issue microcontrollers where load operations pipeline only when grouped together, we group the same kind of independent instructions. As for superscalar processors where multiple execution ports execute at the same time, interleaving different kinds of independent instructions is the way to go. Commonly, there are several instructions that can be dispatched to several execution ports, and also several instructions that are dispatched to a limited subset of the execution ports. For an instruction that can be dispatched to execution ports $\{\texttt{p0}, \dots, \texttt{pn}\}$, we call it a type-$(\texttt{p0|}\cdots\texttt{|pn})$ instruction. If the instruction is decoded into several μops that are dispatched to sets of execution ports $\{\texttt{p00}, \dots, \texttt{p0n}\}, \{\texttt{p10}, \dots, \texttt{p1n}\}, \dots,$ we call the instruction a type-$((\texttt{p00|}\cdots\texttt{|p0n})\,(\texttt{p10|}\cdots\texttt{|p1n})\cdots)$ instruction.

Identifying the execution port with the heaviest workload. We first identify the execution ports with the heaviest workload and schedule the instructions such that instructions that can be dispatched to other ports are dispatched to other ports. For example, suppose we have several type-(`p0`) instructions `I0` and type-(`p0|p1`) instructions `I1`. We identify port `p0` as the one with the heaviest workload. It is advisable to schedule the instructions as $\texttt{I0}, \texttt{I1}, \texttt{I0}, \texttt{I1}, \dots$. Experiments show that `I1`'s have a high chance to be dispatched to port `p1`.

Scheduling a loop. We can generalize the instruction scheduling to more than two execution ports. Below we explain the instruction scheduling of a loop. Typically, the loop body of a loop starts with memory loads and ends with memory stores. Between the loads and stores, there are several arithmetic instructions. Assume that we wish to loop for a loop body l times. We unroll the loop and start moving the instructions to the previous loop bodies. Starting from the 1st loop body (we start counting at 0th), we move the load instructions to the region with the arithmetic instructions of the previous loop body, and arithmetic instructions to the region with the store instructions of the previous loop body. After moving all the instructions, we start interleaving the instructions in the same region. At the beginning, we have a region with load instructions followed by a region with arithmetic and load instructions interleaved. After that, there are $l-1$ regions with load, arithmetic, and store instructions interleaved. At the end, there is a region with arithmetic and store instructions interleaved followed by a region with store instructions. See Figure 10.1 for an illustration.

Figure 10.1: Instruction scheduling of a loop. Red rectangles stand for strings of load instructions, yellow rectangles stand for strings of arithmetic instructions, and purple rectangles stand for strings of store instructions.



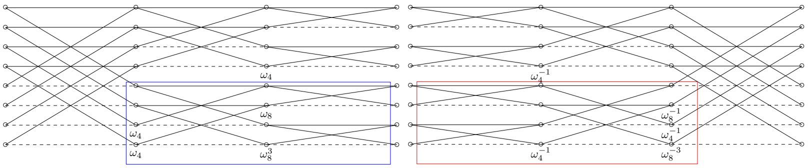
10.2 Isomorphisms

10.2.1 Radix-2 Cooley–Tukey FFT

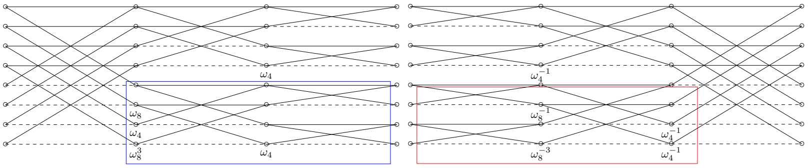
Cooley–Tukey FFT is arguably the commonly used FFT. This section reviews the register pressure for layer-merging and the range analysis.

Register pressure during layer-merging. Following Section 10.1.3, for a positive integer l, an l-layer merge of radix-2 butterflies loads 2^l coefficients and

$2^l - 1$ twiddle factors, applies an l-layer computation, and stores the resulting 2^l coefficients to memory. Figure 10.2 demonstrates the computation flows of 3-layer-merged radix-2 Cooley–Tukey and Gentleman–Sande butterflies for NTT and iNTT. Notice that the memory usage of twiddle factors varies between the chosen modular arithmetic and the coefficient ring. Nevertheless, the register pressure for holding the coefficients gives an upper bound of l. Table 10.3 summarizes the maximum l defining an l-layer merge of radix-2 Cooley–Tukey butterflies based on the register pressure for holding coefficients. In practice, the l-layer-mergings are attainable except for constant-time 32-bit Cooley–Tukey FFT in Armv7-M. The reason is that there is only one processor, Cortex-M3, implementing the Armv7-M without the support of DSP extension, and on Cortex-M3, the long multiplications cannot be used while computing the secret data with constant-time computations. Therefore, one has to resort to software emulations of 32-bit modular multiplications with fairly high register pressure (cf. Algorithms 9.4 and 9.11).



(a) CT for NTT over $x^8 - 1$ and over $x^4 + 1$.
(b) CT for iNTT over $x^8 - 1$ and over $x^4 + 1$.



(c) GS for NTT over $x^8 - 1$ and over $x^4 + 1$.
(d) GS for iNTT over $x^8 - 1$ and over $x^4 + 1$.

Figure 10.2: CT and GS butterflies over $x^8 - 1$ and $x^4 + 1$. ω_n is defined as $\omega^{8/n}$ for ω a principal 8th root of unity. Adapted from [ACC+21].

Table 10.3: Layer-merging of Cooley–Tukey FFT with 16-bit and 32-bit modular multiplications.

ISA/Extension	# reg.	Max. l	Attainable
16-bit Modular Multiplication			
Armv7-M	16	3	✓(constant)
Armv7E-M	16	3	✓(constant)
Armv7-A Neon	16	3	✓(constant)
Armv8-A Neon	32	4	✓(constant)
AVX2	16	3	✓(constant)
AVX-512	32	4	✓(constant)
32-bit Modular Multiplication			
Armv7-M (Cortex-M3)	16	3	✗(constant) / ✓(variable)
Armv7E-M	16	3	✓(constant)
Armv7-A Neon	16	3	✓(constant)
Armv8-A Neon	32	4	✓(constant)
AVX2	16	3	✓(constant)
AVX-512	32	4	✓(constant)

Range analysis. We review the following traditional range analysis arguing the absence of overflows during the Cooley–Tukey FFT computations with precision $\log_2 \mathtt{R}$. Suppose the result of a modular multiplication has an absolute value bounded by θq for a positive real number θ, and we have inputs with absolute values bounded a. After an l-layer radix-2 butterfly, the absolute values of the coefficients are increased by θq so we have coefficients with absolute values bounded by $l\theta q + a$. As long as $l\theta q + a < \frac{\mathtt{R}}{2}$, there are no overflows. For a fixed modulus q, this implies any modular multiplications with quality bounded by $\frac{\mathtt{R}-2a}{2lq}$ suffice. And for a fixed quality θ, the associated modular multiplication can be used for computing over a modulus bounded by $\frac{\mathtt{R}-2a}{2l\theta}$.

10.2.2 Radix-2 Bruun's FFT

We review the implementations of radix-2 Bruun's FFT built upon the factorization of power-of-two cyclotomic polynomials into trinomials. Define

$\mathbf{Bruun}_{\alpha,\beta}$ as follows:

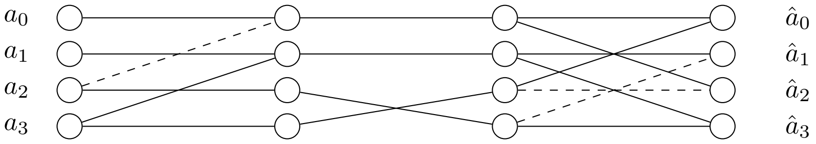
$$\mathbf{Bruun}_{\alpha,\beta}:\begin{cases}\frac{R[x]}{\langle x^4+(2\beta-\alpha^2)x^2+\beta^2\rangle} & \to \frac{R[x]}{\langle x^2+\alpha x+\beta\rangle}\times\frac{R[x]}{\langle x^2-\alpha x+\beta\rangle}\\ a_0+a_1x+a_2x^2+a_3x^3 & \mapsto ((\hat{a}_0+\hat{a}_1x),(\hat{a}_2+\hat{a}_3x))\end{cases}$$

where

$$\begin{cases}(\hat{a}_0,\hat{a}_1)= & \left(a_0-\beta a_2+\alpha\beta a_3, a_1+(\alpha^2-\beta)a_3-\alpha a_2\right),\\ (\hat{a}_2,\hat{a}_3)= & \left(a_0-\beta a_2-\alpha\beta a_3, a_1+(\alpha^2-\beta)a_3+\alpha a_2\right).\end{cases}$$

We compute $(a_0-\beta a_2, a_1+(\alpha^2-\beta)a_3, \alpha a_2, \alpha\beta a_3)$, swap the last two values implicitly, and apply a pair of addition and subtraction (cf. Figure 10.3).

Figure 10.3: Bruun's butterfly. $(\hat{a}_0,\hat{a}_1,\hat{a}_2,\hat{a}_3)=\mathbf{Bruun}_{\alpha,\beta}(a_0,a_1,a_2,a_3)$.



The inverse follows similarly. Define $2\mathbf{Bruun}^{-1}_{\alpha,\beta}$ as follows:

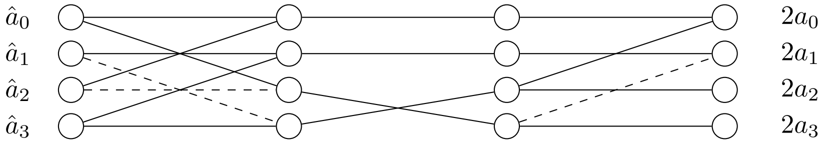
$$2\mathbf{Bruun}^{-1}_{\alpha,\beta}:\begin{cases}\frac{R[x]}{\langle x^2+\alpha x+\beta\rangle}\times\frac{R[x]}{\langle x^2-\alpha x+\beta\rangle} & \to \frac{R[x]}{\langle x^4+(2\beta-\alpha^2)x^2+\beta^2\rangle}\\ ((\hat{a}_0+\hat{a}_1x),(\hat{a}_2+\hat{a}_3x)) & \mapsto 2a_0+2a_1x+2a_2x^2+2a_3x^3\end{cases}$$

where

$$\begin{cases}2(a_0,a_1)= & (\hat{a}_0+\hat{a}_2+(\hat{a}_3-\hat{a}_1)\,\alpha^{-1}\beta, \hat{a}_1+\hat{a}_3-(\hat{a}_0-\hat{a}_2)\,\alpha^{-1}\beta^{-1}\left(\alpha^2-\beta\right)),\\ 2(a_2,a_3)= & ((\hat{a}_3-\hat{a}_1)\,\alpha^{-1},(\hat{a}_0-\hat{a}_2)\,\alpha^{-1}\beta^{-1}).\end{cases}$$

We compute $(\hat{a}_0+\hat{a}_2, \hat{a}_1+\hat{a}_3, \hat{a}_0-\hat{a}_2, \hat{a}_3-\hat{a}_1)$, swap the last two values implicitly, multiply the constants $\alpha^{-1}, \beta, \alpha^{-1}\beta^{-1}$, and $\left(\alpha^2-\beta\right)$, and apply a pair of addition and subtraction (cf. Figure 10.4). The scaling by two is postponed to the end of the computation. Both $\mathbf{Bruun}_{\alpha,\beta}$ and $2\mathbf{Bruun}^{-1}_{\alpha,\beta}$ take four multiplications.

Figure 10.4: Bruun's Inverse butterfly. $(2a_0, 2a_1, 2a_2, 2a_3) = 2\mathbf{Bruun}^{-1}_{\alpha,\beta}(\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3)$.



Special cases of radix-2 isomorphisms. We illustrate the following special cases of radix-2 isomorphisms.

$\mathbf{Bruun}_{\sqrt{2},1}$**:** The initial split of $x^{2^k}+1$ is $\mathbf{Bruun}_{\sqrt{2},1}$. Since $\beta = \alpha^2 - \beta = 1$, we only need two multiplications by $\sqrt{2}$.

$\mathbf{Bruun}_{\alpha,\pm 1}$**:** We avoid multiplying with $\beta = \pm 1$ in $\mathbf{Bruun}_{\alpha,\pm 1}$ and $2\mathbf{Bruun}^{-1}_{\alpha,\pm 1}$.

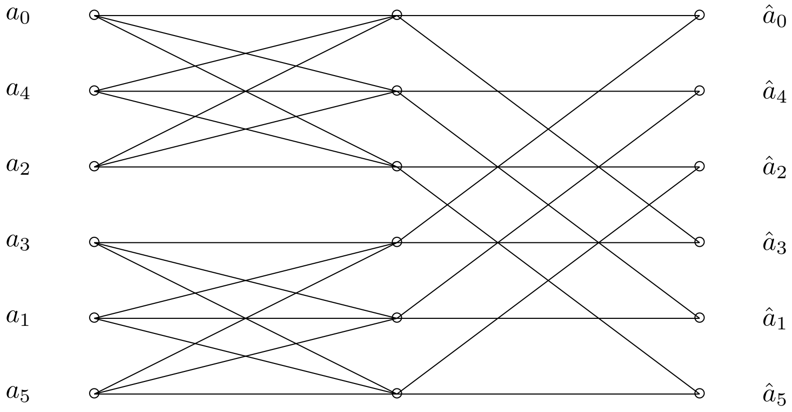
$\mathbf{Bruun}_{\alpha,\frac{\alpha^2}{2}}$**:** We save no multiplications, but only use two constants α and $\frac{\alpha^2}{2}$ instead of four. It is used in the split of $x^{2^k} + \omega_r^{2^k i}$ for an odd r.

Register pressure during layer-merging. As indicated in Figures 10.3 and 10.4, a layer of radix-2 Bruun's FFT over a size-n polynomial is defined on four coefficients distanced apart by $\frac{n}{4}$ coefficients. This implies that for a positive integer l, we need 2^{l+1} coefficients to determine an l-layer merge of radix-2 computation.

10.2.3 Good–Thomas FFT

For Good–Thomas FFT, the primary difference with Cooley–Tukey FFT is about the saving of arithmetic while moving to a different radix at the cost of permuting the input and output coefficients. We can further remove the cost of permutations by permuting on-the-fly while computing a layer of merged computations. Take $R[x]/\langle x^6 - 1\rangle \cong \prod_i R[x]/\langle x - \omega_6^i\rangle$ as an example. We know that $P^{-1}_{6:(14)}\left(\mathcal{F}_{-1} \otimes \mathcal{F}_{\omega_6^4}\right) P_{6:(14)}$. Concretely, we load the inputs while permuting with $P_{6:(14)}$, apply a merged layer computation implementing $\mathcal{F}_{-1} \otimes \mathcal{F}_{\omega_6^4}$, and store the results while permuting with $P^{-1}_{6:(14)}$. See Figure 10.5 for an illustration.

Figure 10.5: Good–Thomas FFT for $R[x]/\langle x^6 - 1\rangle \cong \prod_i R[x]/\langle x - \omega_6^i\rangle$.



10.2.4 Rader's FFT

Let p be an odd prime. Rader-p computes $\mathcal{F}_{\omega_p}$ by computing a size-$(p-1)$ cyclic convolution with linearly number of additions during pre-processing and post-processing, and truncated Rader-p truncates the computation the cyclotomic part $\Phi_p(x)$ of $x^p - 1$.

Rader-p. Define $\texttt{Trapezoid}_n$ as the following $n \times (n+1)$ matrix:

$$\begin{pmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & 0 & 1 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & 0 & 0 & \ddots & 1 & 0 \\ 1 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

We rewrite $\mathcal{F}_{\omega_p}$ as follows:

$$\mathcal{F}_{\omega_p} = \quad \texttt{Trapezoid}_p \left(I_2 \oplus \begin{pmatrix} \omega_p & \omega_p^2 & \cdots & \omega_p^{p-2} & \omega_p^{p-1} \\ \omega_p^2 & \omega_p^4 & \cdots & \omega_p^{p-4} & \omega_p^{p-2} \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ \omega_p^{p-2} & \omega_p^{p-4} & \cdots & \omega_p^4 & \omega_p^2 \\ \omega_p^{p-1} & \omega_p^{p-2} & \cdots & \omega_p^2 & \omega_p \end{pmatrix} \right)$$
$$\left(I_1 \oplus \texttt{Trapezoid}_{p-1}^t\right)$$

where $\texttt{Trapezoid}^t_{p-1}$ and $\texttt{Trapezoid}_p$ amount to linearly number of additions. Rader-p FFT further rewrites as follows:

$$\begin{pmatrix} \omega_p & \omega_p^2 & \cdots & \omega_p^{p-2} & \omega_p^{p-1} \\ \omega_p^2 & \omega_p^4 & \cdots & \omega_p^{p-4} & \omega_p^{p-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \omega_p^{p-2} & \omega_p^{p-4} & \cdots & \omega_p^4 & \omega_p^2 \\ \omega_p^{p-1} & \omega_p^{p-2} & \cdots & \omega_p^2 & \omega_p \end{pmatrix} = \pi_{\texttt{I}} \texttt{Convol} \left(\pi_{\texttt{L}} \left(\omega_p, \dots, \omega_p^{p-1} \right) \right) \pi_{\texttt{R}}$$

for $\pi_{\texttt{I}}, \pi_{\texttt{L}}, \pi_{\texttt{R}}$ permutation matrices of dimension $(p-1) \times (p-1)$ and `Convol` a matrix converting a size-$(p-1)$ vector to the matrix representation of its convolution map. See Algorithm 10.1 for an illustration.

Algorithm 10.1 Pseudocode of Rader-p implementing $R[x]/\langle x^p - 1 \rangle \cong \prod_{i=0}^{p-1} R[x]/\langle x - \omega_p^i \rangle$.

```
Input: $\boldsymbol{a}(x) = \sum_{i=0}^{p-1} a_i x^i$.
Output: $\left( \boldsymbol{a}(1), \boldsymbol{a}(\omega_p), \dots, \boldsymbol{a}\left(\omega_p^{p-1}\right) \right)$.
$\texttt{a[0-}(p-1)\texttt{]} = (a_0, \dots, a_{p-1})$.
$\texttt{c[0]} = \sum_{i=1}^{p-1} \texttt{a[}i\texttt{]}$.                                   ▷ Apply $\texttt{Trapezoid}^t_{p-1}$.
$\texttt{b[1-}(p-1)\texttt{]} = \pi_{\texttt{L}} \left( \omega_p, \dots, \omega_p^{p-1} \right)$.                 ▷ Apply $\pi_{\texttt{L}}$
$\texttt{a[1-}(p-1)\texttt{]} = \pi_{\texttt{R}} \left( \texttt{a[1-}(p-1)\texttt{]} \right)$.                 ▷ Apply $\pi_{\texttt{R}}$.
$\texttt{c[1-}(p-1)\texttt{]} = \texttt{b[1-}(p-1)\texttt{]} \cdot \texttt{a[1-}(p-1)\texttt{]} \bmod \left( x^{p-1} - 1 \right)$.
$\texttt{c[1-}(p-1)\texttt{]} = \pi_{\texttt{I}} \left( \texttt{c[1-}(p-1)\texttt{]} \right)$.                 ▷ Apply $\pi_{\texttt{I}}$.
$\texttt{c[0-}(p-1)\texttt{]} = \texttt{c[0-}(p-1)\texttt{]} + (\texttt{a[0]}, \dots, \texttt{a[0]})$.     ▷ Apply $\texttt{Trapezoid}_p$.
```

Optimizing the number of additions in Rader-p. We illustrate [PBT+24, Section III-B]'s ideas on optimizing $\texttt{Trapezoid}^t_{p-1}$ and $\texttt{Trapeziod}_p$. Suppose we find an $\mathcal{F}$ implementing the following:

$$\begin{aligned} \texttt{Convol} \left(\pi_{\texttt{L}} \left(\omega_p, \dots, \omega_p^{p-1} \right) \right) = \quad & (p-1)\mathcal{F}^{-1} \\ & \texttt{ScaledMul}_{\mathcal{F}} \left(\mathcal{F} \left(\pi_{\texttt{L}} \left(\omega_p, \dots, \omega_p^{p-1} \right) \right) \right) \mathcal{F} \end{aligned}$$

where $\texttt{ScaledMul}_{\mathcal{F}}$ is the ring multiplication in the image of $\mathcal{F}$ with proper scaling on the output. For $\texttt{Trapezoid}^t_{p-1}$, the sum is already computed after applying $\mathcal{F}$ so we simply retain the desired sum while applying $\texttt{ScaledMul}_{\mathcal{F}}$ and remove $\texttt{Trapezoid}^t_{p-1}$. As for $\texttt{Trapezoid}_p$, we similarly apply the inversion turning $p-1$ additions into 1 addition right after $\texttt{ScaledMul}_{\mathcal{F}}$. We summarize the overall computation as follows:

$$\begin{aligned}\mathcal{F}_{\omega_p} = \quad & \left(I_1 \oplus \pi_{\mathtt{I}}(p-1)\mathcal{F}^{-1}\right)\mathcal{T}'_p\left(I_2 \oplus \mathtt{ScaledMul}_{\mathcal{F}}\left(\mathcal{F}\left(\pi_{\mathtt{L}}\left(\omega_p, \dots, \omega_p^{p-1}\right)\right)\right)\right)\\ & \mathcal{T}_p\left(I_1 \oplus \mathcal{F}\pi_{\mathtt{R}}\right)\end{aligned}$$

for $\mathcal{T}_p$ the following $(p+1) \times p$ matrix

$$\mathcal{T}_p = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots \\ 0 & \vdots & \ddots & \ddots & \ddots \\ 0 & 0 & \ddots & \ddots & 1 \end{pmatrix}$$

and $\mathcal{T}'_p$ the following $p \times (p+1)$ matrix

$$\mathcal{T}'_p = \begin{pmatrix} 1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots \\ 0 & 0 & \ddots & \ddots & \ddots & \ddots \end{pmatrix}.$$

Computing odd-size DFT with Good–Thomas and Rader's FFTs. We review how to apply an odd-size DFT with Good–Thomas and Rader's FFTs [AHY22, Section 3.1.2]. Let r be an odd integer and $\mathcal{F}_{\omega_r}$ be the DFT we wish to apply. Whenever r contains more than one prime factor, we apply Good–Thomas turning $\mathcal{F}_{\omega_r}$ into a tensor product of $\mathcal{F}_{\omega_{r_i}}$'s with $r_i \perp r_j$ for $i \neq j$. For each $\mathcal{F}_{\omega_{r_i}}$, if r_i is a prime power $p_i^{d_i}$ with $d_i > 1$, we apply Winograd's FFT turning it into a size-$p_i^{d_i-1}(p_i-1)$ cyclic convolution. As $p_i^{d_i-1} \perp (p_i-1)$, we apply Good–Thomas. Notice that $p_i - 1$ is even, and we can also apply Good–Thomas if $p_i - 1$ contains some odd factors. The Good–Thomas and Winograd permutations terminate when all $p_i - 1$ are powers of two. Therefore, we are left with a tensor product of $\mathcal{F}_{\omega_{p_i}}$'s with p_i's Fermat primes. For each $\mathcal{F}_{\omega_{p_i}}$ with Fermat prime p_i, we apply Rader-p_i turning it into a size-(p_i-1) cyclic convolution. Since p_i is a Fermat prime, the convolution has size a power of two. If $\mathcal{F}_{\omega_{p_i-1}}$ is defined, we apply radix-2 Cooley–Tukey FFT for the size-(p_i-1) cyclic convolution. In the worst case, $\mathcal{F}_{-1}$ is always defined and we apply it whenever possible.

Truncated Rader-p. With the same notation, we apply truncated Rader-p for $R[x]/\langle\Phi_p(x)\rangle \cong \prod_{i=0}^{p-2} R[x]/\left\langle x - \omega_p^{i+1}\right\rangle$ as shown in Algorithm 10.2.

Algorithm 10.2 Pseudocode of truncated Rader-p over $R[x]/\langle\Phi_p(x)\rangle$.

```
Input: $\boldsymbol{a}(x) = \sum_{i=0}^{p-2}$.
Output: $\left(\boldsymbol{a}(\omega_p), \boldsymbol{a}\left(\omega_p^2\right), \dots, \boldsymbol{a}\left(\omega_p^{p-1}\right)\right)$.
a[0-$(p-2)$] $= (a_0, \dots, a_{p-2})$.
b[0-$(p-2)$] $= \pi_{\mathtt{L}}\left(\omega_p, \dots, \omega_p^{p-1}\right)$.    ▷ Apply $\pi_{\mathtt{L}}$.
a[0-$(p-2)$] $= \pi_{\mathtt{R}}$(a[0-$(p-2)$]).    ▷ Apply $\pi_{\mathtt{R}}$.
c[0-$(p-2)$] $=$ b[0-$(p-2)$] $\cdot$ a[0-$(p-2)$] mod $\left(x^{p-1} - 1\right)$.
c[0-$(p-2)$] $= \pi_{\mathtt{I}}$(c[0-$(p-2)$]).    ▷ Apply $\pi_{\mathtt{I}}$.
c[0-$(p-2)$] $=$ c[0-$(p-2)$] $\cdot \left(\omega_p, \dots, \omega_p^{p-1}\right) \in R^{p-1}$.    ▷ This step is often
    merged with other multiplications.
```

10.3 Monomorphisms That Are Not Isomorphisms

10.3.1 Karatsuba, Toom–Cook, Striding Karatsuba, Striding Toom–Cook

For Karatsuba and Toom–Cook, we can apply some memory optimizations. For simplicity, we illustrate the ideas with Toom–Cook.

Memory optimizations for Toom–Cook interpolation. Let $k|n$. Recall that $\mathbf{TC}_{(2k-1)\times k}$ transforms as follows

$$R[x]_{<n} \cong \left(\frac{R[x]}{\left\langle x^{\frac{n}{k}} - y\right\rangle}\right)[y]_{<k} \hookrightarrow \frac{\left(R[x]_{<\frac{2n}{k}-1}\right)[y]}{\left\langle \prod_{i=0}^{2k-2}(y - s_i)\right\rangle} \cong \prod_{i=0}^{2k-2} \frac{\left(R[x]_{<\frac{2n}{k}-1}\right)[y]}{\langle y - s_i\rangle}$$

and results in polynomial multiplications in $R[x]_{<\frac{2n}{k}-1}$. For the inversion, a straightforward approach is to invert $\cong$ and $\hookrightarrow$ separately [KRS19, NG21], incurring at least two layers of memory operations. To reduce the memory operations, we can instead hold the results of $\mathbf{TC}^{-1}_{(2k-1)\times(2k-1)}$ in registers, we invert $\hookrightarrow$ before storing them into memory. See Figure 10.6 for an illustration of applying $\mathbf{TC}^{-1}_{3\times 3}$ to $R[x]_{<6}$.

Figure 10.6: Interpolation of $\mathbf{TC}_{3\times 3}^{-1}$ for polynomials in $R[x]_{<6}$. We accumulate $\mathbf{TC}_{3\times 3}^{-1}$ and $\mathbf{TC}_{3\times 3}^{-1}$, and store the results to memory.

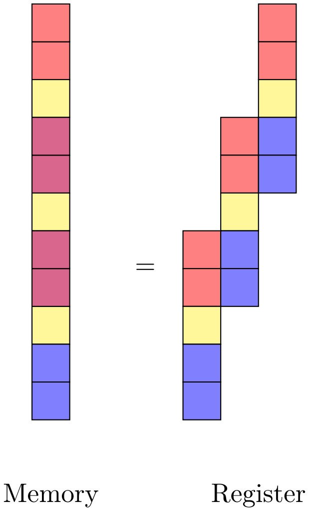
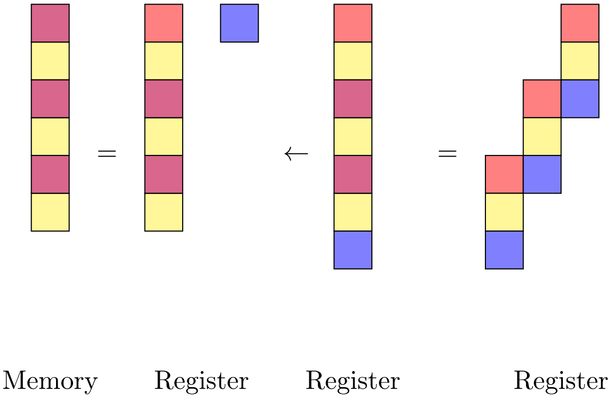


Figure 10.7: Interpolation of striding $\mathbf{TC}_{3\times 3}^{-1}$ for polynomials in $R[x]/\langle x^6+1\rangle$. We accumulate $\mathbf{TC}_{3\times 3}^{-1}$ and $\mathbf{TC}_{3\times 3}^{-1}$, and store the results to memory.



Toom–Cook vs striding Toom–Cook. As mentioned in Section 6.4.2, striding exploits the structural property of the polynomial ring. We illustrate the memory and register pressure of striding Toom–Cook with $R[x]/\left\langle x^6+1\right\rangle$. Striding $\mathbf{TC}_{3\times 2}$ works as follows:

$$\frac{R[x]}{\langle x^6+1\rangle} \cong \frac{\left(R[y]/\left\langle y^3+1\right\rangle\right)[x]}{\langle x^2-y\rangle} \hookrightarrow \left(\frac{R[y]}{\langle y^3+1\rangle}\right)[x]_{<3}.$$

During the inversion, we have three size-3 polynomials from $R[y]/\left\langle y^3+1\right\rangle$ instead of three size-5 polynomials from $R[x]_{<5}$ in the Toom–Cook case. Therefore, the register pressure is reduced. See Figure 10.7 for an illustration.

10.3.2 Radix-2 Schönhage's and Nussbaumer's FFTs

We go through some implementation strategies of radix-2 Schönhage and Nussbaumer. In Schönhage and Nussbaumer, multiplications by twiddle factors are negacyclic shifts. For scalar implementations, negacyclic shifts are implemented in the on-the-fly fashion – we load the coefficients from the desired position and remove the shifting, and replace the follow-up add-sub pairs with sub-add pairs in the butterflies and skip the negations. For vector implementations, there are two lines of approaches.

- Extraction: If there is an instruction extracting consecutive bytes from the concatenation of two vectors, we negate of the operands and extract the desired bytes. Algorithm 10.3 illustrates the Armv8-A Neon implementation holding 16-bit elements [HLY24].
- Unaligned load and store: If there is no extract instructions, we can implement the negacyclic shifts with unaligned memory operations [BBCT22].

Algorithm 10.3 Negacyclic shift $i = 1, \ldots, 7$ coefficients of polynomials in $R[x]/\left\langle x^8+1\right\rangle$ with R a coefficient ring fit into a 16-bit halfword with Armv8-A Neon [HLY24].

Input:
Output:

```
ldr      q0, [src]
neg.8H   v1,     v0
ext.16B  v0,     v0, v1, #2$i$
str      q0, [des]
```

Algorithm 10.4 Negacyclic shift $i = 1, \ldots, 15$ coefficients of polynomials in $R[x]/\langle x^{16} + 1 \rangle$ with R a coefficient ring fit into a 16-bit halfword in AVX2 [BBCT22].

Input:
Output:

```
vpxor          zero,          zero, zero
vmovdqu      (%src),            v0
vmovdqu   $2i$(%src),           v1
vpsubw           v0,          zero,   v0
vmovdqu          v0, $(16-2i)$(%des)
vmovdqu          v1,          (%des)
```

Register pressure during layer-merging. In Schönhage and Nussbaumer, twiddle-factor multiplications are negacyclic shifts, and when we move to the next layer, there are doubly many negacyclic shifts where two registers are involved in each butterfly. This implies a factor of 4 blow-up, and we need 2^{2l-1} registers for an l-layer merge in radix-2 cyclic Schönhage and Nussbaumer, and 2^{2l} registers in radix-2 negacyclic Schönhage.

Optimizing the layer merging during pre- and post-processing in Nussbaumer. We can also reduce the memory access during the extension of the coefficient rings. We skip the explicit replacement of relations and the initial butterflies, and modify the memory load in the follow-up butterflies accordingly. For the converse replacement, we also merge it with the last series of butterflies.

Part III

Applications to Lattice-Based Cryptosystems

Chapter 11

Benchmarking Methodology

11.1 Cortex-M3 and Cortex-M4

11.1.1 Benchmarking Environment

Cortex-M3. This thesis benchmarks the performance of Armv7-M implementations on the board `nucleo-f207zg` with the `stm32f207zg` Cortex-M3 core, 128 KB RAM, and 1 MB flash. The board is clocked at 30 MHz for consistency with the literature. Nevertheless, the clock cycles will remain almost the same if benchmarked at the the maximum frequency 120 MHz with 0-wait state [STM20].

Cortex-M4. As for Armv7E-M implementations, this thesis benchmarks the implementations on the board `stm32f4discovery` with the `stm32f407vg` Cortex-M4 core, FPU, 1 MB flash memory, and 128 KB RAM, and 64 KB core-coupled memory (CCM). The frequency is fixed to 24 MHz with 0-wait state for consistency with the literature. The board could be clocked at the maximum frequency 168 MHz at the cost of additional cycles for fetching instructions.

Software requirements. As Cortex-M3 and Cortex-M4 are microcontrollers, we need to compile the source code with a cross-compiler. We cross-compile the source code with `arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10.3-2021.10) 10.3.1 20210824 (release)`. For the cross-compilation, we also rely on the external firmware library `libopencm3`[1]. For completeness, we also include the linker scripts `stm32f207zg.ld` for Cortex-M3 and `stm32f4discovery.ld` for Cortex-M4 generated by the script `genlink.py`[2] For flashing the binaries to the boards, we choose `openocd 0.12.0`[3] with the configuration file `nucleo-f2.cfg` for `nucleo-f207zg`, and `stlink v1.8.0`[4] for `stm32f4discovery`. For reading data from the broad, this thesis provides Python scripts compatible with `Python 3.13.3`.

11.1.2 Common Functions

For the lattice-based cryptosystems covered by this thesis, many of the subroutines are shared. We use the same pseudorandom number derived generator from ChaCha20 [Ber08a], sorting network from `djbsort`[5] with the optimizations by [AHY22], and `sha2` from `supercop-20220506`[6].

11.1.3 Lattice-Based Cryptosystems

For the lattice-based cryptosystems, we benchmark the performance cycles of (i) cryptographic operations such as key generation, encapsulation, decapsulation, signature generation, and signature verification; (ii) cryptographic hash functions such as SHA2 and SHA3 family; and (iii) polynomial arithmetic such as polynomial multiplication, inner product, and matrix-vector multiplication. We also test for correctness and compatibility of testvectors, provide Python scripts for processing the output from the board.

11.1.4 Comments on Other Performance Metric

Cycle count reduction is the primary optimization goal in this thesis. However, cycle count is not the only performance metric for microcontroller in real-world deployment. We also take memory consumption and code size into account. This thesis does not further optimize for memory consumption as long as the resulting binary files are small enough for flashing to the board. As for the code size, the binaries benefit from the periodic structure of the fast homomorphisms and have reasonable code size.

---

[1] `https://github.com/libopencm3/libopencm3`. Commit `5e7dc5d092e52bbfbb8b5929e2097732e1b7f81c`.

[2] `https://github.com/libopencm3/libopencm3/blob/master/scripts/genlink.py`.

[3] `https://openocd.org/`.

[4] `https://github.com/stlink-org/stlink/releases`.

[5] `https://sorting.cr.yp.to/`.

[6] `crypto_hash/sha512/ref/hash.c` and `crypto_hashblocks/sha512/m3/inner.S`.

11.2 Cortex-A72 and Firestorm

11.2.1 Benchmarking Environment

This thesis benchmarks the performance of Armv8-A Neon-optimized implementations on two platforms – Cortex-A72 and Firestorm. We outline below the benchmarking environment.

Cortex-A72. For Cortex-A72, we benchmark on a raspberry pi 4 clocked at 1.5 GHz. We follow the work [BHK$^+$21] and access the cycle counter with the instruction `mrs`. The source code is compiled with `Ubuntu GCC 12.3.0-1ubuntu1 23.04`.

Firestorm. Firestorm is the performance core on Apple M1 series and this thesis benchmarks its performance on an Apple M1 Pro running at 3.23 GHz. We access the cycle counters through macOS private APIs[7]. The programs are compiled with `Homebrew GCC 13.3.0`.

11.2.2 Common Functions

For the common functions, we use the ChaCha20 implementation from [CCHY24] for the pseudorandom number generator, Keccak permutation from [NG21] for SHA3 family on Cortex-A72 and with SHA3 instructions by Bas Westerbaan[8] on Firestorm; sorting network from [CCHY24]; and the reference implementation of SHA2 from `supercop-20220506`[9].

11.2.3 Lattice-Based Cryptosystems

For the lattice-based cryptosystems, we benchmark the medium performance cycles of cryptographic operations and polynomial arithmetic on Cortex-A72, and average performance cycles on Firestorm. We also test for correctness and compatibility of testvectors.

---

[7] Adapted from `https://gist.github.com/ibireme/173517c208c7dc333ba962c1f0d67d12`.

[8] `https://github.com/bwesterb/armed-keccak`.

[9] `crypto_hash/sha512/ref/`.

11.3 Haswell

11.3.1 Benchmarking Environment

This thesis benchmarks the performance of x86-64 AVX2-optimized implementations on Haswell, in particular, Intel(R) Core(TM) i7-4770K with the nomial frequency 3.5 GHz. The source code are compiled with `gcc (Debian 12.2.0-14) 12.2.0`. Turbo Boost and hyperthreading are turned off during benchmarking. We access the cycle counter with the instruction `rdtsc` for the benchmark.

11.3.2 Common Functions

For the common functions, we use ChaCha20 from `supercop-20220506`[10] for the pseudorandom number generation, AVX2-optimized Keccak permutation by Keccak team[11] shipped with the Kyber repository[12] for the SHA3 family; sorting network from `supercop-20220506`[13]; and SHA2 from `supercop-20220506`[14]

11.3.3 Lattice-Based Cryptosystems

For the lattice-based cryptosystems, we benchmark the average performance cycles of cryptographic operations and polynomial arithmetic. We also test for correctness and compatibility of testvectors.

---

[10] `crypto_stream/chacha20/e/ref/`.

[11] https://github.com/XKCP/XKCP.

[12] https://github.com/pq-crystals/kyber. Commit `bb5be790e1b17400d107f934fc00cd313cdc323a`

[13] `crypto_sort/int32/avx2/`.

[14] `crypto_hash/sha512/ref/`.

Chapter 12

Dilithium

12.1 Specification

Dilithium is a lattice-based digital signature [ABD+20a] selected as a winner in the NIST PQC Standardization, and is based on the hardness of M-LWE and M-SIS. We briefly review the algebraic structures and parameters relevant to this thesis and refer to [ABD+20a] for the specification. Let $q = 2^{23} - 2^{13} + 1$ be a prime, $n = 256$, and $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. Parameters k, ℓ determine the module $R_q^{k\times\ell}$ over R_q, parameter η determines the universe of the coefficients of the secret polynomials, parameter τ determines the number of non-zero entries in the challenge polynomial, and parameter d determines the drop bits in the rounding. See Table 12.1 for a summary of the parameters relevant to this thesis.

Table 12.1: Dilithium parameters [ABD+20a] relevant to this work.

Parameter set	k	ℓ	η	τ	d	# rep.
`dilithium2`	4	4	2	39	13	4.25
`dilithium3`	6	5	4	49	13	5.1
`dilithium5`	8	7	2	60	13	3.85

Key generation. In the key generation, we concatenate the input bit string ξ with the byte representation of k, ℓ, and hash the resulting bit string into three parts: the public seed $\rho \in \{0,1\}^{256}$, the secret seed $\rho' \in \{0,1\}^{512}$, and the remaining bit string $K \in \{0,1\}^{256}$. The public seed ρ is expanded into a $k \times \ell$

public matrix $\hat{\mathbf{A}}$ over $\mathsf{NTT}(R_q)$, and the seed ρ' is expanded into a size-ℓ secret vector $\mathbf{s}_1$ and a size-k secret vector $\mathbf{s}_2$ over R_q. Notice that the matrix $\hat{\mathbf{A}}$ is expanded in the NTT domain. We then compute $\mathbf{t} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\mathsf{NTT}(\mathbf{s}_1)\right) + \mathbf{s}_2$, and component-wisely round $\mathbf{t}$ into $\mathbf{t}_0$ and $\mathbf{t}_1$ based on the value of d. The public key pk is defined as $(\rho, \mathbf{t}_1)$ and the secret key sk is defined as $(\rho, K, \mathcal{H}pk, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$. In this thesis, we are interested in the computation of

$$\mathbf{t} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\mathsf{NTT}(\mathbf{s}_1)\right) + \mathbf{s}_2.$$

See Algorithm 12.1 for an illustration.

Algorithm 12.1 Dilithium key generation.

```
Input: $\xi \in \{0,1\}^{256}$.
Output:
    $\begin{cases} pk &= \texttt{pkEncode}(\rho, \mathbf{t}_1). \\ sk &= \texttt{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0). \end{cases}$
$(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} \leftarrow \mathcal{H}(\xi||k||\ell)$
$(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^l \times S_\eta^k \leftarrow \texttt{ExpandS}(\rho')$
$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(\rho)$
$\mathbf{t} \leftarrow \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\mathsf{NTT}(\mathbf{s}_1)\right) + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \texttt{Power2Round}(\mathbf{t})$
$pk \leftarrow \texttt{pkEncode}(\rho, \mathbf{t}_1)$
$tr \in \{0,1\}^{512} \leftarrow \mathcal{H}(pk)$
$sk \leftarrow \texttt{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
```

Signature generation. In the signature generation, we construct a bit string $\rho'' \in \{0,1\}^{512}$ by hashing parts of the secret key, the message, and the input randomness. We then expand ρ'' into a vector $\mathbf{y}$ and compute $\mathbf{w} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\mathsf{NTT}(\mathbf{y})\right)$. From $\mathbf{w}$, we construct the polynomial $\mathbf{w}_1$ consisting of the high bits of $\mathbf{w}$, hash it to $\tilde{c}$ along with a hash of parts of the secret key, and sample the challenge polynomial c from $\tilde{c}$. From c, we compute vectors of products $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$, $\mathbf{r} = c\mathbf{s}_2$, and $\mathbf{h}' = c\mathbf{t}_0$, and compute the hints $\mathbf{h}$ from $\mathbf{h}', \mathbf{w}, \mathbf{r}$. We then test if $\mathbf{z}$, $\mathbf{r}$, $\mathbf{h}'$, and $\mathbf{h}$ meet certain conditions. If all the tests pass, we define the signature as $(\tilde{c}, \mathbf{z}, \mathbf{h})$. In this thesis, we are interested in the computations of

$$\mathbf{w} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\mathsf{NTT}(\mathbf{y})\right), \mathbf{z} = \mathbf{y} + c\mathbf{s}_1, \mathbf{r} = c\mathbf{s}_2, \mathbf{h}' = c\mathbf{t}_0.$$

See Algorithm 12.2 for an illustration.

Algorithm 12.2 Dilithium signature generation.

```
Input:
    $\begin{cases} sk & = \texttt{skEncode}\left(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0\right). \\ m & \in \{0,1\}^*. \\ rnd & \in \{0,1\}^{256}. \end{cases}$
Output: $\sigma = \texttt{sigEncode}\left(\tilde{c}, \mathbf{z}, \mathbf{h}\right).$
$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) = \texttt{skDecode}(sk)$
$\hat{\mathbf{A}} \in R_q^{k\times\ell} = \texttt{ExpandA}(\rho)$
$\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}\left((tr||m\right)$
$\rho'' \in \{0,1\}^{512} \leftarrow \mathcal{H}(K||rnd||\mu)$
$\kappa \leftarrow 0$
$\hat{\mathbf{s}}_1 \leftarrow \mathsf{NTT}\left(\mathbf{s}_1\right)$; $\hat{\mathbf{s}}_2 \leftarrow \mathsf{NTT}\left(\mathbf{s}_2\right)$; $\hat{\mathbf{t}}_0 \leftarrow \mathsf{NTT}\left(\mathbf{t}_0\right)$
$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
while $(\mathbf{z}, \mathbf{h}) = \perp$ do
    $\mathbf{y} \in S_{\gamma_1-1}^{\ell} \leftarrow \texttt{ExpandMask}(\rho'', \kappa)$
    $\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$; $\mathbf{w} \leftarrow \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{y}}\right)$
    $\mathbf{w}_1 \leftarrow \texttt{HighBits}(\mathbf{w})$; $\tilde{c} \in \{0,1\}^{256} \leftarrow \mathcal{H}(\mu||\mathbf{w}_1)$; $c \leftarrow \mathcal{H}_B(\tilde{c})$
    $\hat{c} \leftarrow \mathsf{NTT}(c)$; $\mathbf{z} \leftarrow \mathbf{y} + \mathsf{NTT}^{-1}\left(\hat{c}\hat{\mathbf{s}}_1\right)$; $\mathbf{r} \leftarrow \mathsf{NTT}^{-1}\left(\hat{c}\hat{\mathbf{s}}_2\right)$
    $\mathbf{r}_0 \leftarrow \texttt{LowBits}(\mathbf{w} - \mathbf{r})$
    if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ or $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ then
        $(\mathbf{z}, \mathbf{h}) = \perp$
    else
        $\mathbf{h}' \leftarrow \mathsf{NTT}^{-1}\left(\hat{c}\hat{\mathbf{t}}_0\right)$
        $\mathbf{h} \leftarrow \texttt{MakeHint}\left(-\mathbf{h}', \mathbf{w} - \mathbf{r} + \mathbf{h}'\right)$
        if $\|\mathbf{h}'\|_\infty \geq \gamma_2$ or $\#$ 1's in $\mathbf{h} > \omega$ then
            $(\mathbf{z}, \mathbf{h}) = \perp$
    $\kappa \leftarrow \kappa + 1$
$\sigma \leftarrow \texttt{sigEncode}\left(\tilde{c}, \mathbf{z}, \mathbf{h}\right)$
```

Signature verification. As for the signature verification with the signature $(\tilde{c}, \mathbf{z}, \mathbf{h})$, we compute $\mathbf{w}'_{\texttt{Approx}} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{z}} - \mathsf{NTT}\left(2^d c\right)\mathsf{NTT}\left(\mathbf{t}_1\right)\right)$ and test if $\mathbf{w}'_{\texttt{Approx}}$ meets a certain condition. In this thesis, we are interested in computing $\mathbf{w}'_{\texttt{Approx}}$. See Algorithm 12.3 for an illustration.

Algorithm 12.3 Dilithium signature verification.

```
Input:
    $\begin{cases} pk & = \texttt{pkEncode}\left(\rho, \mathbf{t}_1\right). \\ m & \in \{0,1\}^*. \\ \sigma & = \texttt{sigEncode}\left(\tilde{c}, \mathbf{z}, \mathbf{h}\right). \end{cases}$
Output: $\top$ or $\bot$.
$(\rho, \mathbf{t}_1) \leftarrow \texttt{pkDecode}(pk)$
$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \texttt{sigDecode}(\sigma)$
if $\mathbf{h} = \bot$ then
    return $\bot$
if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ then
    return $\bot$
$\hat{\mathbf{A}} \in R_q^{k \times \ell} \leftarrow \texttt{ExpandA}(\rho)$
$\hat{\mathbf{z}} \leftarrow \mathsf{NTT}(\mathbf{z})$
$tr \in \{0,1\}^{512} \leftarrow \mathcal{H}(pk)$
$\mu \in \{0,1\}^{512} \leftarrow \mathcal{H}(tr||m)$
$c \leftarrow \mathcal{H}_B(\tilde{c})$
$\mathbf{w}'_{\texttt{Approx}} \leftarrow \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{z}} - \mathsf{NTT}\left(2^d c\right)\mathsf{NTT}\left(\mathbf{t}_1\right)\right)$
$\mathbf{w}'_1 \leftarrow \texttt{UseHint}(\mathbf{h}, \mathbf{w}'_{\texttt{Approx}})$
if $\tilde{c} \neq \mathcal{H}(\mu||\mathbf{w}'_1)$ then
    return $\bot$
return $\top$
```

12.2 Optimization Guide for Polynomial Arithmetic

12.2.1 Dilithium NTTand NTT^{-1}, and MV

Dilithium NTTand NTT^{-1}. For NTTand NTT^{-1}defined on R_q (those colored in blue in Algorithms 12.1, 12.2, and 12.3), we only need to decide the 32-bit modular arithmetic as the transformation is already determined at the design phase. Since $q = 8380417$ is a prime, we need to design efficient 32-bit modular arithmetic. On platforms with efficient 32-bit high or long multiplication instructions, we apply 32-bit Montgomery and Barrett multiplications. On the other hand, if there are no native 32-bit high and long multiplication instructions or if 32-bit high and long multiplication instructions cannot be used, one has to emulate the high and long multiplications with 32-bit low multiplication instructions. In this case, one should choose the approximate variant of Barrett multiplication while trading the efficiency of the high product with the accuracy. After determining the approximate high multiplication in Barrett multiplication, the remaining part is range analysis (cf. Section 10.2.1). Table 12.2 summarizes the upper bounds of the modulus q defining an 8-layer Cooley–Tukey FFT with several qualities θs of modular multiplications without overflows.

Table 12.2: Relations between the quality θ of variants of Barrett multiplications and the modulus q for an 8-layer radix-2 Cooley–Tukey FFT.

Variant	θ	Upper bound of q	Upper bound of $\log_2 q$
Standard	0.75	306 783 378	28.1926
Floor	1.75	143 165 576	27.0931
Half-approx.	2.75	93 368 854	26.4763
Approximate	3.75	69 273 666	26.0458

Matrix-vector multiplication. For the matrix-vector multiplication, after deploying the Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$, the next step is to implement the base multiplications with improved accumulation. This is commonly implemented with Montgomery multiplication.

12.2.2 Challenge Polynomial Multiplication

In the signature generation, we have to multiply polynomials $\mathbf{s}_1, \mathbf{s}_2$, and $\mathbf{t}_0$ by the challenge polynomial c. As the coefficients of $c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$ are bounded by $\eta\tau, \eta\tau$, and $2^{d-1}\tau$ in absolute value, we can compute them over a modulus larger than $2\eta\tau, 2\eta\tau$, and $2^d\tau$, respectively. Table 12.3 summarizes the lower bounds of the modulus allowing us computing $c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$ with modulus switching. For $c\mathbf{s}_1$ and $c\mathbf{s}_2$, we can choose a 16-bit modulus defining a radix-2 NTT and compute $c\mathbf{s}_1$ and $c\mathbf{s}_2$ over the newly chosen modulus. Furthermore, for `dilithium2` and `dilithium5`, we can choose $257 = 2^{2^3} + 1$, the 4th Fermat prime, as the modulus, turning several twiddle-factor multiplications into shifts [AHKS22]. As for $c\mathbf{t}_0$, we compute with 32-bit NTT/iNTT if 32-bit modular arithmetic is efficient, and with NTT/iNTT defined over two 16-bit modulus otherwise. We can also employ Nussbaumer's and Schönhage's FFTs over $\mathbb{Z}_{2^k}$, completely avoiding modular multiplications during the transformation. Table 12.4 summarizes the design space for the challenge polynomial multiplications.

Table 12.3: Lower bounds on the modulus implementing $c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$ in Dilithium with modulus switching.

Parameter set	$c\mathbf{s}_1$	$c\mathbf{s}_2$	$c\mathbf{t}_0$
`dilithium2`	156	156	319488
`dilithium3`	392	392	401408
`dilithium5`	240	240	491520

Table 12.4: Overview of the design space of challenge polynomial multiplications $c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$ in `dilithium2`, `dilithium3`, and `dilithium5`. Sch. stands for Schönhage's FFT and Nuss. stands for Nussbaumer's FFT.

Parameter set	Operation	16-bit NTT	16-bit FNT	Sch./Nuss.
`dilithium2`	$c\mathbf{s}_1$	✓	✓	✓
	$c\mathbf{s}_2$	✓	✓	✓
	$c\mathbf{t}_0$	✗	✗	✓(32-bit)
`dilithium3`	$c\mathbf{s}_1$	✓	✗	✓(32-bit)
	$c\mathbf{s}_2$	✓	✗	✓(32-bit)
	$c\mathbf{t}_0$	✗	✗	✓(32-bit)
`dilithium5`	$c\mathbf{s}_1$	✓	✓	✓
	$c\mathbf{s}_2$	✓	✓	✓
	$c\mathbf{t}_0$	✗	✗	✓(32-bit)

12.3 Reviewing and Improving Cortex-M3 Implementations

For the Cortex-M3 implementations of Dilithium, this thesis reviews and benchmarks the implementations by [GKS20, HAZ+24, HKS24].

12.3.1 Dilithium `NTT`, $\mathtt{NTT}^{-1}$, and MV

Dilithium `NTT`/$\mathtt{NTT}^{-1}$. For Dilithium `NTT` and $\mathtt{NTT}^{-1}$ on Cortex-M3, the critical part is the efficient modular multiplication implementing the twiddle-factor multiplications. Since long multiplication instructions `{u, s}{mul, mla}l` take variable-time, the resulting `NTT` and $\mathtt{NTT}^{-1}$ take variable-time and can only be used in computing public data. For Dilithium `NTT` and $\mathtt{NTT}^{-1}$ with secret inputs, we have to deploy constant-time modular multiplications. [GKS20] proposed Montgomery multiplication with `s{mul, mla}l` (cf. Algorithm 9.1) for the variable-time `NTT`/$\mathtt{NTT}^{-1}$, and Montgomery multiplication with `{mul, mla}` (cf. Algorithm 9.4) for the constant-time `NTT`/$\mathtt{NTT}^{-1}$, and [HKS24] generalized Barrett multiplication by relaxing the accuracy of the high multiplication. The relaxation enables one to trade the performance cycles of the emulation of high multiplication with its accuracy, and leads to significant improvement for modular multiplication (cf. Table 9.8) when precomputation of one of the input operands is free. See Table 12.5 for a summary of the performance. As shown in Table 12.5, Barrett-based `NTT`/$\mathtt{NTT}^{-1}$ significantly outperform the Montgomery-based `NTT`/$\mathtt{NTT}^{-1}$ as the expensive long multiplications and their emulations are replaced with a single long multiplication in the variable-time case and an efficient approximation of the high multiplication in the constant-time case.

Table 12.5: Performance cycles of Dilithium `NTT`s and $\mathtt{NTT}^{-1}$s on Cortex-M3.

Operation	Work	Constant-time	Variable-time
`NTT`	[GKS20]*	33 079	19 228
	[HKS24]*	**20 258**	**15 126**
$\mathtt{NTT}^{-1}$	[GKS20]*	36 658	20 871
	[HKS24]*	**24 201**	**17 725**

*Benchmark of this thesis.

Matrix-vector multiplications. For the Dilithium constant-/variable-time matrix-vector multiplications, [GKS20] deployed the Montgomery-based constant-/variable-time `NTT`/$\mathtt{NTT}^{-1}$, and [HKS24] improved the performance with Barrett-based constant-/variable-time `NTT`/$\mathtt{NTT}^{-1}$. Another bottleneck is the base multiplication after the `NTT`. In [GKS20], they computed the base multiplications one at a time, and [HKS24] followed [CHK+21] and accumulated several products prior to the modular reductions in the inner products. See Table 12.6 for a summary of the performance. As shown in Table 12.6, [HKS24] significantly improved the inner products in the matrix-vector multiplications. Along with Barrett-based $\mathsf{NTT}/\mathsf{NTT}^{-1}$, [HKS24] significantly improved the matrix-vector multiplications compared to prior art by [GKS20].

Table 12.6: Performance cycles of Dilithium matrix-vector multiplications and size-ℓ inner products in the NTT domain on Cortex-M3.

Operation	Work	Constant-time	Variable-time
Matrix-vector multiplication			
`dilithium2`	[GKS20]*	427 669	253 901
	[HKS24]*	**276 146**	**202 214**
`dilithium3`	[GKS20]*	662 714	394 492
	[HKS24]*	**421 483**	**307 401**
`dilithium5`	[GKS20]*	1 043 297	618 586
	[HKS24]*	**642 955**	**468 439**
Inner product in NTT domain			
`dilithium2`	[GKS20]*	35 553	21 724
	[HKS24]*	**19 035**	**13 894**
`dilithium3`	[GKS20]*	45 560	27 302
	[HKS24]*	**23 903**	**17 240**
`dilithium5`	[GKS20]*	64 069	38 490
	[HKS24]*	**33 119**	**23 904**

*Benchmark of this thesis.

12.3.2 Challenge Polynomial Multiplication

Polynomial multiplications for $c\mathbf{s}_1$ and $c\mathbf{s}_2$. For the challenge polynomial multiplications $c\mathbf{s}_1$ and $c\mathbf{s}_2$, we have several choices. [AHKS22] proposed the uses of 16-bit NTT over 769 for `dilithium3` and 16-bit FNT over 257 for `dilithium2` and `dilithium5` on Cortex-M4. Following [AHKS22]'s proposal, [HAZ+24] implemented the 16-bit Plantard-based NTT over 769 for `dilithium3` on Cortex-M3, and [HKS24] implemented the 16-bit FNT over 257 for `dilithium2` and `dilithium5` on Cortex-M3. Table 12.7 summarizes the performance of 16-bit Montgomery-based NTT by [ACC+21], 16-bit Plantard-based NTT by [HAZ+24], and 16-bit FNT over 257 by [HKS24].

Table 12.7: Performance cycles of polynomial multiplications with 16-bit arithmetic precision on Cortex-M3. The numbers of [ACC$^+$21] in this table were reported in [Hwa22, Table 9.10].

Work	[ACC$^+$21]	[HAZ$^+$24]	[HKS24]
Coefficient ring	$\mathbb{Z}_{3329}$	$\mathbb{Z}_{769}$	$\mathbb{Z}_{257}$
Approach	Montgomery	Plantard	FNT
NTT	8 688	7 830	**7 252**
Mul.	5 987	3 989	**2 835**
iNTT	9 553	8 543	**7 667**

Table 12.8: Performance cycles of polynomial multiplications with 32-bit arithmetic precision on Cortex-M3. The total cycles of polynomial multiplications are obtained by summing up all the rows in the building block.

Work	[ACC$^+$21]	[HAZ$^+$24]	[HKS24]
Coefficient ring	$\prod_{i=0,1} \mathbb{Z}_{q_i}$	$\prod_{i=0,1} \mathbb{Z}_{q_i}$	$\mathbb{Z}_{2^{\leq 24}}$
Approach	Montgomery	Plantard	Nussbaumer
Building block			
NTT/Hom-M	16 774	15 626	15 716
NTT/Hom-V	16 774	15 626	7 993
Mul./BiHom	11 933	8 061	10 317
iNTT/Hom-I	23 721	20 772	10 767
Polynomial multiplication			
Total cycles	69 202	60 085	**44 793**
Ratio of mul./BiHom	17.24%	13.42%	23.03%
Memory (bytes)	1 536	2 048	14 848

Polynomial multiplications for $c\mathbf{t}_0$. For the challenge polynomial multiplications $c\mathbf{t}_0$, we also have several choices. Following [ACC$^+$21], computing over 16-bit NTT-friendly moduli is faster than computing over a 32-bit NTT-friendly modulus on Cortex-M3. Their Montgomery-based implementations can already be used in Dilithium on Cortex-M3. [HAZ$^+$24] replaced the

16-bit Montgomery-based NTT with 16-bit Plantard-based NTT, but compared against the 32-bit NTT approach of [GKS20] even though the 16-bit Montgomery-based NTT of [ACC+21] can already be used in Dilithium on Cortex-M3. [HKS24] applied Nussbaumer followed by Toeplitz-TC over $\mathbb{Z}_{2^k}$. Since we are computing entirely over $\mathbb{Z}_{2^k}$ in Nussbaumer+Toeplitz-TC, the Hom-M and Hom-I phases are significantly improved at the cost of slightly slower BiHom phase, and the performance of polynomial multiplication is improved. See Table 12.8 for a summary of the performance.

Table 12.9: Performance cycles of $c\mathbf{s}_1, c\mathbf{s}_2, c\mathbf{t}_0$ in the rejection loop of signature generation in Dilithium on Cortex-M3.

Parameter set	Operation	[HAZ+24]	[HKS24]
`dilithium2`	$c\mathbf{s}_1$	50 128	**41 672**
	$c\mathbf{s}_2$	50 128	**41 672**
	$c\mathbf{t}_0$	115 332	**87 969**
`dilithium3`	$c\mathbf{s}_1$	62 660	-
	$c\mathbf{s}_2$	75 192	-
	$c\mathbf{t}_0$	172 998	-
`dilithium5`	$c\mathbf{s}_1$	87 724	**72 891**
	$c\mathbf{s}_2$	100 256	**83 296**
	$c\mathbf{t}_0$	230 664	-

$c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$. This thesis benchmarks [HKS24]'s FNT for $c\mathbf{s}_1$ and $c\mathbf{s}_2$ in `dilithium2` and `dilithium5`, and their Nussbaumer+Toeplitz-TC over $\mathbb{Z}_{2^k}$ for $c\mathbf{t}_0$ in `dilithium2`. Notice that Toeplitz-TC should be used in $c\mathbf{t}_0$ as we only need to apply the expensive Hom-M only once. If we switch to other symmetric approaches such as Toom–Cook, then we have to apply the expensive interpolation several times. In theory, one can also apply the Nussbaumer+Toeplitz-TC approach to `dilithium3` and `dilithium5`, but in reality the approach consumes a lot of memory and cannot be deployed to `dilithium3` and `dilithium5` on Cortex-M3. Table 12.9 summarizes the performance of $c\mathbf{s}_1$, $c\mathbf{s}_2$, and $c\mathbf{t}_0$ where the transformation cost of c is excluded. The numbers of [HAZ+24] are projections based on Tables 12.7 and 12.8 since their benchmark setup was flawed. [HAZ+24] counted the transformation cost of c twice while reporting numbers of $c\mathbf{s}_1$ and $c\mathbf{s}_2$, but this should be counted only once even if the cost of c is counted.

12.3.3 Scheme

For the overall performance of Dilithium on Cortex-M3, this thesis benchmarks the Barrett-based constant-/variable-time Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$ [HKS24], the improved accumulation for the base multiplications in the matrix-vector multiplications [CHK$^+$21, HKS24], the 16-bit FNT over 257 for $c\mathbf{s}_1$ and $c\mathbf{s}_2$ in `dilithium2` and `dilithium5` [AHKS22, HKS24], and the Nussbaumer+Toeplitz-TC for $c\mathbf{t}_0$ in `dilithium2` [HKS24]. See Table 12.10 for a summary of the performance.

Table 12.10: Performance cycles of Dilithium on Cortex-M3.

Parameter set	Work	**K**	**S**	**V**
`dilithium2`	[HAZ$^+$24]*	1 766k	5 887k	1 606k
	[HKS24]*	**1 542k**	**4 766k**	**1 519k**
`dilithium3`	[HAZ$^+$24]*	2 943k	9 527k	2 670k
	[HKS24]*	**2 668k**	**8 218k**	**2 531k**
`dilithium5`	[HAZ$^+$24]*	4 925k	19 389k	4 536k
	[HKS24]*	**4 449k**	**17 666k**	**4 305k**

*Benchmark of this thesis.

12.4 Reviewing and Improving Cortex-M4 Implementations

For the Cortex-M4 implementations of Dilithium, this thesis reviews and benchmarks the implementations by [AHKS22, HAZ$^+$24].

12.4.1 Dilithium NTT, NTT^{-1}, and MV

For the Dilithium NTT, NTT^{-1}, and matrix-vector multiplications, the fastest approach is the Montgomery-based approach – we exploit the powerful 1-cycle long multiplication instructions `s{mul, mla}l` on Cortex-M4. The state-of-the-art implementations by [AHKS22] incorporated several optimizations such as the uses of floating-point registers as “low-latency cache” [ACC$^+$20] and the improved accumulation for the base multiplications [CHK$^+$21]. See Tables 12.11 and 12.12 for a summary of the performance.

Table 12.11: Performance cycles of Dilithium NTTs and NTT^{-1}s on Cortex-M4.

Work	NTT	NTT^{-1}
[AHKS22]*	8 098	8 418

*Benchmark of this thesis.

Table 12.12: Performance cycles of Dilithium matrix-vector multiplications and size-ℓ inner products in the NTT domain on Cortex-M4.

Parameter set	Work	Inner prod.	MV
`dilithium2`	[AHKS22]*	9 528	103 675
`dilithium3`	[AHKS22]*	12 038	162 550
`dilithium5`	[AHKS22]*	17 068	259 630

*Benchmark of this thesis.

12.4.2 Challenge Polynomial Multiplication

For the challenge polynomial multiplications $c\mathbf{s}_1, c\mathbf{s}_2$, [AHKS22] proposed 16-bit FNT over 257 for `dilithium2` and `dilithium5` and 16-bit Montgomery-based NTT over 769 for `dilithium3`. [HAZ+24] later replaced the 16-bit Montgomery-based NTT with 16-bit Plantard-based NTT for `dilithium3`. See Table 12.13 for a summary of the performance.

Table 12.13: Performance cycles of challenge polynomial multiplications $c\mathbf{s}_1$ and $c\mathbf{s}_2$.

Operation	Work	`dilithium2`	`dilithium3`	`dilithium5`
$c\mathbf{s}_1$	[AHKS22]*	28 827	-	48 257
	[HAZ+24]*	-	41 304	-
$c\mathbf{s}_2$	[AHKS22]*	26 914	-	53 782
	[HAZ+24]*	-	45 460	-

*Benchmark of this thesis.

12.4.3 Scheme

For the overall performance of Dilithium on Cortex-M4, this thesis benchmarks the state-of-the-art implementations [AHKS22, HAZ+24]. See Table 12.14 for a summary of the performance.

Table 12.14: Performance cycles of Dilithium on Cortex-M4.

Parameter set	Work	**K**	**S**	**V**
`dilithium2`	[AHKS22]*	1 386k	3 366k	1 381k
`dilithium3`	[AHKS22, HAZ+24]*	2 439k	5 584k	2 340k
`dilithium5`	[AHKS22]*	5 685k	29 733k	9 200k

*Benchmark of this thesis.

12.5 Reviewing and Improving Armv8-A Neon Implementations

For the Armv8-A Neon implementations of Dilithium, this thesis reviews the implementations by [BHK+21, ABKK23], improves the Dilithium NTT/NTT^{-1}, and benchmarks the implementations by [BHK+21] and the improved implementations by this thesis.

12.5.1 Dilithium NTT, NTT^{-1}, and MV

Table 12.15: Performance cycles of Dilithium NTTs and NTT^{-1}s with Armv8-A Neon on Cortex-A72 and Firestorm.

Parameter set	Work	Cortex-A72	Firestorm
NTT	[BHK+21]*	2 246	478
	[ABKK23]	**1 766**	-
	This thesis	1 811	**423**
NTT^{-1}	[BHK+21]*	2 825	582
	This thesis	**2 535**	**525**

*Benchmark of this thesis.

Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$. For the Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$ with Armv8-A Neon, [NG21] proposed Montgomery multiplication with long multiplication instructions `s{mul, mla, mls}l{, 2}`, and [BHK+21] introduced the Barrett multiplication built upon the high multiplication instruction `sqrdmulh` (cf. Algorithm 9.19). [ABKK23] improved the instruction scheduling with an automatic tool, and this thesis improved the instruction scheduling with macroized instruction interleaving. See Table 12.15 for a summary of the performance.

Matrix-vector multiplications. As for the matrix-vector multiplications, this thesis integrates the Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$ with macroized instruction interleaving into [BHK+21]. See Table 12.16 for a summary of the performance.

Table 12.16: Performance cycles of Dilithium matrix-vector multiplications and inner products in the NTT domain with Armv8-A Neon on Cortex-A72 and Firestorm.

Parameter set	Work	Cortex-A72	Firestorm
Inner product			
`dilithium2`	[BHK+21]*	1 206	251
`dilithium3`	[BHK+21]*	1 380	276
`dilithium5`	[BHK+21]*	1 815	386
Matrix vector multiplication			
`dilithium2`	[BHK+21]*	27 111	5 362
	This thesis	**23 231**	**4 900**
`dilithium3`	[BHK+21]*	40 182	7 819
	This thesis	**36 429**	**7 186**
`dilithium5`	[BHK+21]*	56 110	11 696
	This thesis	**52 189**	**108 141**

*Benchmark of this thesis.

12.5.2 Scheme

For the overall performance of Dilithium with Armv8-A Neon, this thesis integrates the improved Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$ in the matrix-vector multiplications into [BHK+21]. See Table 12.17 for a summary of the performance.

Table 12.17: Performance cycles of Dilithium with Armv8-A Neon on Cortex-A72 and Firestorm.

Parameter set	Work	**K**	**S**	**V**
Cortex-A72				
`dilithium2`	[BHK+21]*	281 216	816 325	281 627
	This thesis	**279 315**	**793 311**	**278 943**
`dilithium3`	[BHK+21]*	536 860	1 002 610	469 008
	This thesis	**533 555**	**979 416**	**461 352**
`dilithium5`	[BHK+21]*	822 482	1 744 381	817 199
	This thesis	**818 711**	**1 695 664**	**796 541**
Firestorm				
`dilithium2`	[BHK+21]*	78 645	244 480	79 290
	This thesis	**78 259**	**239 964**	**78 621**
`dilithium3`	[BHK+21]*	168 150	372 363	122 214
	This thesis	**167 467**	**364 616**	**121 261**
`dilithium5`	[BHK+21]*	207 481	454 827	198 826
	This thesis	**206 668**	**446 332**	**197 962**

*Benchmark of this thesis.

12.6 Reviewing AVX2 Implementations

For the AVX2-optimized implementations of Dilithium, this thesis reviews and benchmarks the implementations by [ABD+20a].

12.6.1 Dilithium NTT, NTT^{-1}, and MV

Dilithium NTT/NTT^{-1}. For the AVX2-optimized Dilithium NTT/NTT^{-1}, the main difference with the Neon-optimized implementation is the absence of 32-bit high multiplication instructions. Therefore, one has to resort to the long multiplication instruction `vpmuldq` multiplying the lower 32-bit elements of each 64-bit elements and returning the 64-bit elements. Notice that `vpmuldq` is the only instruction in AVX2 computing the signed long multiplications and the upper 32-bit elements have to be moved to lower 32-bit parts prior to calling `vpmuldq`. If one transfers the upper 32-bit to the lower parts with shift instructions `vpsrld` and `vpsrad`, then the instructions might compete with the multiplication instruction `vpmuld`. We can instead call the floating-point duplication `vmovshdup` (cf. Algorithm 9.21). See Table 12.18 for a summary of the performance.

Matrix-vector multiplications. For the Dilithium matrix-vector multiplications, we can similarly apply the improved accumulation for the base multiplications [ABD+20a]. See Table 12.18 for a summary of the performance.

Table 12.18: Performance cycles of Dilithium $\mathsf{NTT}/\mathsf{NTT}^{-1}$, matrix-vector multiplications, and size-ℓ inner products in NTT domain with AVX2 on Haswell.

Parameter set	Work	NTT	NTT^{-1}	Inner prod.	MV
`dilithium2`	[ABD+20a]*	1 205	1 328	593	12 887
`dilithium3`	[ABD+20a]*	1 205	1 328	679	18 613
`dilithium5`	[ABD+20a]*	1 205	1 328	887	26 836

*Benchmark of this thesis.

12.6.2 Scheme

See Table 12.19 for a summary of the performance.

Table 12.19: Performance cycles of Dilithium with AVX2 on Haswell.

Parameter set	Work	**K**	**S**	**V**
`dilithium2`	[ABD+20a]*	98 168	315 244	107 895
`dilithium3`	[ABD+20a]*	167 775	505 007	173 433
`dilithium5`	[ABD+20a]*	265 611	604 048	270 500

*Benchmark of this thesis.

Chapter 13

Kyber

13.1 Specification

Kyber is a lattice-based KEM [ABD$^+$20b] selected as a winner in the NIST PQC Standardization, and is based on the hardness of M-LWE. For simplicity, we go the underlying PKE scheme and refer to [ABD$^+$20b] for the full specification of the KEM. Let $q = 3329, n = 256, k, \eta_1, \eta_2, d_u$, and d_v be integers where k, η_1, η_2, d_u, and d_v vary between parameter sets. Parameters q and n determine the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, and parameter k determines the module $R_q^{k\times k}$ over R_q. Parameters η_1 and η_2 determine the centered binomial distributions for the secret values, and parameters d_u and d_v determine the compressions of ciphertexts. See Table 13.1 for a summary of the parameter sets.

Table 13.1: Kyber parameter sets.

Parameter set	q	n	k	η_1	η_2	d_u	d_v
`kyber512`	3329	256	2	3	2	10	4
`kyber768`	3329	256	3	2	2	10	4
`kyber1024`	3329	256	4	2	2	11	5

Key generation. In the key generation, we generate a matrix $\mathbf{A} \in R_q^{k\times k}$ from the seed ρ and sample vectors $\mathbf{s}, \mathbf{e} \in R_q^k$ from the centered binomial distribution with η_1 as the parameter in both vectors and the seed σ, compute the vector $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} \in R_q^k$. Conceptually, $(\mathbf{t}, \rho)$ forms the public key and $\mathbf{s}$

forms the secret key. Let NTT be the following isomorphism:

$$\mathsf{NTT} : R_q = \frac{\mathbb{Z}_q[x]}{\langle x^n + 1 \rangle} \cong \prod_{0 \leq i < 128} \frac{\mathbb{Z}_q[x]}{\left\langle x^2 - \omega_{256}^{2\mathtt{rev}_{2:7}(i)+1} \right\rangle}.$$

We define $\hat{\mathbf{t}} := \mathsf{NTT}(\mathbf{t}), \hat{\mathbf{A}} := \mathsf{NTT}(\mathbf{A}), \hat{\mathbf{s}} := \mathsf{NTT}(\mathbf{s})$, and $\hat{\mathbf{e}} := \mathsf{NTT}(\mathbf{e})$. Concretely, we sample $\hat{\mathbf{A}}$ from the seed ρ, compute $\hat{\mathbf{s}}$ and $\hat{\mathbf{e}}$ by applying NTT, and compute $\hat{\mathbf{t}} = \hat{\mathbf{A}}\hat{\mathbf{s}} + \hat{\mathbf{e}}$. The public key is defined as $(\hat{\mathbf{t}}, \rho)$ and the secret key is defined as $\hat{\mathbf{s}}$. See Algorithm 13.1 for an illustration.

Algorithm 13.1 Kyber PKE key generation.

Input: Randomness $rnd \in \{0,1\}^{256}$.

Output:

$$\begin{cases} \text{Public key} & pk & = \mathtt{pkEncode}\left(\hat{\mathbf{t}}, \rho\right). \\ \text{Secret key} & sk & = \mathtt{skEncode}\left(\hat{\mathbf{s}}\right). \end{cases}$$

```
$(\rho, \sigma) \leftarrow \mathcal{H}(rnd || k)$
$\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \mathtt{ExpandA}(\rho)$
$(\mathbf{s}, \mathbf{e}) \in R_q^k \times R_q^k \leftarrow \mathtt{SampleSE}_{\eta_1, \eta_1}(\sigma)$
$\hat{\mathbf{s}} \leftarrow \mathsf{NTT}(\mathbf{s})$; $\hat{\mathbf{e}} \leftarrow \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}}\hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk = \mathtt{pkEncode}(\hat{\mathbf{t}}, \rho)$
$sk = \mathtt{skEncode}(\hat{\mathbf{s}})$
```

Encryption. For encryption, we first sample vectors $\mathbf{y}, \mathbf{e}_1 \in R_q^k$, and $e \in R_q$ from the centered binomial distributions with parameters η_1, η_2, and η_2, respectively. After decompressing the message m to a polynomial $\mu \in R_q$, we compute $\hat{\mathbf{y}} = \mathsf{NTT}(\mathbf{y})$, $\mathbf{u} = \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{y}}\right) + \mathbf{e}$, and $v = \mathsf{NTT}^{-1}\left(\hat{\mathbf{t}}^\top \hat{\mathbf{y}}\right) + e_2 + \mu$. The ciphertext ct is defined as the juxtaposition of the compressions of $\mathbf{u}$ and v. See Algorithm 13.2 for an illustration.

Algorithm 13.2 Kyber PKE encryption.

```
Input:
    $\begin{cases} \text{Public key} & pk & = \texttt{pkEncode}\left(\hat{\mathbf{t}}, \rho\right). \\ \text{Message} & m & \in \{0,1\}^{256}. \\ \text{Randomness} & rnd & \in \{0,1\}^{256}. \end{cases}$
Output: Ciphertext $ct = (c_1, c_2)$.
$\left(\hat{\mathbf{t}}, \rho\right) \leftarrow \texttt{pkDecode}(pk)$
$\hat{\mathbf{A}} \in R_q^{k \times k} \leftarrow \texttt{ExpandA}\,(\rho)$
$(\mathbf{y}, \mathbf{e}_1, e_2) \in R_q^k \times R_q^k \times R_q \leftarrow \texttt{SampleYEE}_{\eta_1,\eta_2,\eta_2}\,(rnd)$
$\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{y}}\right) + \mathbf{e}_1$
$\mu \leftarrow \texttt{Decompress}_1\,(\texttt{ByteDecode}_1(m))$
$v \leftarrow \mathsf{NTT}^{-1}\left(\hat{\mathbf{t}}^\top \hat{\mathbf{y}}\right) + e_2 + \mu$
$c_1 \leftarrow \texttt{ByteEncode}_{d_u}\left(\texttt{Compress}_{d_u}(\mathbf{u})\right)$
$c_2 \leftarrow \texttt{ByteEncode}_{d_v}\left(\texttt{Compress}_{d_v}(v)\right)$
$ct \leftarrow (c_1, c_2)$
```

Decryption. For decryption, we decompress the ciphertext ct into a vector $\mathbf{u}' \in R_q^k$ and $v' \in R_q$, compute $w = v' - \mathsf{NTT}^{-1}\left(\hat{\mathbf{s}}^\top \mathsf{NTT}\left(\mathbf{u}'\right)\right)$, and compress the polynomial w into the message m. See Algorithm 13.3 for an illustration.

Algorithm 13.3 Kyber PKE decryption.

```
Input:
    $\begin{cases} \text{Ciphertext} & ct & = (c_1, c_2). \\ \text{Secret key} & sk & = \texttt{skEncode}\left(\hat{\mathbf{s}}\right). \end{cases}$
Output: Message $m \in \{0,1\}^{256}$.
$\mathbf{u}' \leftarrow \texttt{Decompress}_{d_u}\left(\texttt{ByteDecode}_{d_u}(c_1)\right)$
$v' \leftarrow \texttt{Decompress}_{d_v}\left(\texttt{ByteDecode}_{d_v}(c_2)\right)$
$\hat{\mathbf{s}} \leftarrow \texttt{skDecode}(sk)$
$w \leftarrow v' - \mathsf{NTT}^{-1}\left(\hat{\mathbf{s}}^\top \mathsf{NTT}(\mathbf{u}')\right)$
$m \leftarrow \texttt{ByteEncode}_1\left(\texttt{Compress}_1(w)\right)$
```

13.2 Optimization Guide for Polynomial Arithmetic

This section goes through the overall optimization strategies for polynomial arithmetic in Kyber based on author's experience. We go through the NTTs $\mathsf{NTT}(\mathbf{s}), \mathsf{NTT}(\mathbf{e}), \mathsf{NTT}(\mathbf{y})$, and $\mathsf{NTT}(\mathbf{u}')$, iNTTs $\mathsf{NTT}^{-1}\left(\hat{\mathbf{A}}\hat{\mathbf{s}}\right), \mathsf{NTT}^{-1}\left(\hat{\mathbf{t}}^\top\hat{\mathbf{y}}\right)$, and $\mathsf{NTT}^{-1}\left(\hat{\mathbf{s}}^\top\mathsf{NTT}(\mathbf{u}'')\right)$, matrix-vector products $\hat{\mathbf{A}}\hat{\mathbf{s}}$ and $\hat{\mathbf{A}}\hat{\mathbf{y}}$, inner products $\hat{\mathbf{t}}^\top\hat{\mathbf{y}}$ and $\hat{\mathbf{s}}^\top\mathsf{NTT}(\mathbf{u}')$, and compressions $\texttt{Compress}_{d_u}(\mathbf{u})$ and $\texttt{Compress}_{d_v}(v)$ in Algorithms 13.1, 13.2, and 13.3.

13.2.1 NTT and NTT^{-1}

For NTT and NTT^{-1}, since we are asked to compute certain transformations, we only need to decide the 16-bit modular arithmetic in the coefficient ring, the available registers on the target platforms, and the strategies of instruction scheduling. We choose between Montgomery, Barrett, and Plantard modular multiplications based on the analysis in Section 9.2.1.4. Once the modular arithmetic is determined, we next look into the number of available registers, determine the layer-merging strategy as explained in Section 10.2.1, and schedule the instructions (cf. Section 10.1.4).

13.2.2 Matrix-Vector Multiplication and Inner Product

For the inner products, we first look into the availability of long multiplications. If there are long multiplications, we compute the long products, accumulate them, and apply Montgomery reduction. If there are no long multiplications, we apply the same modular multiplication as $\mathsf{NTT}/\mathsf{NTT}^{-1}$. For the matrix-vector multiplication, essentially we compute several inner products with the same vector operand. Since one of the vector operands is the same, we can put more workload on this vector operand and save the overall arithmetic with "asymmetric multiplication" [BHK+21]. Suppose we want to compute the small-dimensional products

$$\begin{cases} (a_{0,0}b_0 + \zeta a_{0,1}b_1) + (a_{0,0}b_1 + a_{0,1}b_0)\,x, \\ (a_{1,0}b_0 + \zeta a_{1,1}b_1) + (a_{1,0}b_1 + a_{1,1}b_0)\,x, \\ (a_{2,0}b_0 + \zeta a_{2,1}b_1) + (a_{2,0}b_1 + a_{2,1}b_0)\,x. \end{cases}$$

We first compute the matrix $\begin{pmatrix} b_0 & \zeta b_1 \\ b_1 & b_0 \end{pmatrix}$ can store it in memory, and rewrite the small-dimensional products as the following matrix-vector products:

$$\begin{pmatrix} b_0 & \zeta b_1 \\ b_1 & b_0 \end{pmatrix} \begin{pmatrix} a_{0,0} \\ a_{0,1} \end{pmatrix}, \begin{pmatrix} b_0 & \zeta b_1 \\ b_1 & b_0 \end{pmatrix} \begin{pmatrix} a_{1,0} \\ a_{1,1} \end{pmatrix}, \begin{pmatrix} b_0 & \zeta b_1 \\ b_1 & b_0 \end{pmatrix} \begin{pmatrix} a_{2,0} \\ a_{2,1} \end{pmatrix}.$$

This allows us to compute the products in $\mathbb{Z}_q[x]/\langle x^2 - \zeta \rangle$ as in $\mathbb{Z}_q[x]/\langle x^2 - 1 \rangle$. Overall, we replace three multiplications by ζ with one multiplication by ζ at the cost of additional memory for storing ζb_1.

13.2.3 Compressions

For the compressions $\texttt{Compress}_d$ with $d = 1, 4, 5, 10, 11$, they follow from Section 9.3.

13.3 Reviewing and Improving Cortex-M3 Implementations

For the Cortex-M3 implementations of Kyber, this thesis reviews the implementations by [ABD+20b, HZZ+24], improves the compressions, and benchmarks the implementations by [ABD+20b, HZZ+24] and the improved implementations by this thesis.

13.3.1 Kyber NTT, NTT^{-1}, MV, and IP

Kyber NTT/NTT^{-1}. For the Kyber NTT/NTT^{-1} on Cortex-M3, the fastest approach is the 16-bit Plantard multiplication using the 32-bit multiplication instructions `mul`/`mla`/`mls` as shown Algorithm 9.14 by [HZZ+24], essentially following the signed 32-bit Plantard multiplication with 64-bit low multiplications by [AMOT22]. Table 13.2 reports the performance of the 16-bit NTT/NTT^{-1} based on the 16-bit Plantard multiplication by [HZZ+24].

Table 13.2: Performance cycles of NTT and NTT^{-1} in Kyber on Cortex-M3.

Work	NTT	NTT^{-1}
[HZZ+24]*	8 031	8 598

*Benchmark of this thesis.

Matrix-vector multiplications and inner products. Built upon the 16-bit Plantard-based `NTT`, $\mathtt{NTT}^{-1}$, and base multiplications in the NTT domain, the matrix-vector multiplications and the inner products are also improved. Since matrix-vector multiplications and inner products are interleaved with the polynomial samplings, their performance are not straightforward to benchmark and [HZZ+24] didn't report the performance of matrix-vector multiplications and inner products. We benchmark the overall cycles and the sampling cycles of the subroutines and subtract them to derive the cycles spent on the matrix-vector multiplications and inner products. See Table 13.3 for a summary of the performance of matrix-vector multiplications and inner products.

Table 13.3: Performance cycles of matrix-vector multiplications, inner products in encryptions, and inner products in decryptions of Kyber on Cortex-M3.

Operation	Work	`kyber512`	`kyber768`	`kyber1024`
MV	[HZZ+24]*	88 172	166 101	270 016
IP(Enc)	[HZZ+24]*	17 377	21 251	25 126
IP(Dec)	[HZZ+24]*	33 333	45 184	57 036

*Benchmark of this thesis.

13.3.2 $\mathtt{Compress}_d$

For the compression functions $\mathtt{Compress}_{\{1,4,5,10,11\}}$, this thesis implements the Barrett-based compressions in C and in Armv7-M assembly (cf. Section 9.3.2). Table 13.4 summarizes the performance of the C implementations by [ABD+20b] with several follow-up updates, the Barrett-based C implementations, and the assembly-optimized Barrett-based implementations. As shown in Table 13.4, the performance of compression functions are already improved after switching the to Barrett-based C implementations. The assembly implementations further optimize the memory operations and packing with barrel shifter.

Table 13.4: Performance cycles of $\texttt{Compress}_d$ in Kyber on Cortex-M3.

Operation	[ABD+20b]* (Ref)	C (This thesis)	Asm. (This thesis)
$\texttt{Compress}_1$	3 558	2 847	**1 557**
$\texttt{Compress}_4$	3 411	1 933	**1 556**
$\texttt{Compress}_5$	3 249	2 315	**1 805**
$\texttt{Compress}_{10}$	5 583	2 764	**1 765**
$\texttt{Compress}_{11}$	5 299	2 868	**1 904**

*Benchmark of this thesis. All the numbers here refer to the implementations of `https://github.com/pq-crystals/kyber/commit/10b478fc3cc4ff6215eb0b6a11bd758bf0929cbd` where the compression functions were updated in the commits `https://github.com/pq-crystals/kyber/commit/dda29cc63af721981ee2c831cf00822e69be3220` and `https://github.com/pq-crystals/kyber/commit/272125f6acc8e8b6850fd68ceb901a660ff48196`.

13.3.3 Scheme

For the overall performance of Kyber on Cortex-M3, this thesis integrates the assembly-optimized Barrett-based compressions to [HAZ+24], and benchmarks the implementations. See Table 13.5 for the summary of the performance.

Table 13.5: Performance cycles of Kyber on Cortex-M3.

Parameter set	Work	**K**	**E**	**D**
`kyber512`	[HAZ+24]*	**447k**	442k	507k
	This thesis	**447k**	**435k**	**498k**
`kyber768`	[HAZ+24]*	**730k**	742k	831k
	This thesis	**730k**	**733k**	**819k**
`kyber1024`	[HAZ+24]*	**1 152k**	1 159k	1 274k
	This thesis	**1 152k**	**1 146k**	**1 260k**

*Benchmark of this thesis.

13.4 Reviewing and Improving Cortex-M4 Implementations

For the Cortex-M4 implementations of Kyber, this thesis reviews the implementations by [ABD+20b, AHKS22, HZZ+22], improves the compressions, and benchmarks the implementations by [ABD+20b, AHKS22, HZZ+22] and the improved implementations by this thesis.

13.4.1 Kyber `NTT`, NTT^{-1}, MV, and IP

On Cortex-M4, the overall strategies of the fastest `NTT`, NTT^{-1}, matrix-vector multiplications, and inner products are very similar to the Cortex-M3 implementation. For the Kyber $\mathsf{NTT}/\mathsf{NTT}^{-1}$, [AHKS22] optimized the 16-bit Montgomery-based approach by [ABCG20] with floating-point registers as "low-latency cache" [ACC+20], faster 16-bit Barrett reductions (already used in [ACC+20] and reported in [AHKS22]), the adoption of Cooley–Tukey FFT for NTT^{-1} [ACC+21], the asymmetric multiplication for the base multiplication [BHK+21], and the improved accumulation for the base multiplication [CHK+21]. [HZZ+22] later proposed the 16-bit Plantard multiplication for the modular multiplications (cf. Algorithm 9.15). Tables 13.6 and 13.7 summarize the performance of `NTT`, NTT^{-1}, matrix-vector multiplications, and inner products.

Table 13.6: Performance cycles of `NTT` and NTT^{-1} in Kyber on Cortex-M4.

Operation	[AHKS22]	[HZZ+22]*
`NTT`	5 992	**4 489**
NTT^{-1}	5 491	**4 665**

*Benchmark of this thesis.

Table 13.7: Performance cycles of matrix-vector multiplications, inner products in encryption, and inner products in decryption of Kyber on Cortex-M4.

Operation	Work	`kyber512`	`kyber768`	`kyber1024`
MV	[AHKS22]*	60 292	114 613	184 557
	[HZZ⁺22]*	**55 212**	**107 797**	**176 728**
IP(Enc)	[AHKS22]*	8 764	10 336	11 906
	[HZZ⁺22]*	**7 937**	**9 509**	**11 078**
IP(Dec)	[AHKS22]*	23 437	32 368	41 297
	[HZZ⁺22]*	**19 655**	**27 205**	**34 753**

*Benchmark of this thesis.

13.4.2 `Compress`$_d$

Table 13.8: Performance cycles of `Compress`$_d$ in Kyber on Cortex-M4.

C		
Operation	[ABD⁺20b]* (Ref)	Barrett (This thesis)
`Compress`$_1$	3 356	**3 041**
`Compress`$_4$	3 177	**2 603**
`Compress`$_5$	3 019	**2 825**
`Compress`$_{10}$	4 044	**3 082**
`Compress`$_{11}$	3 725	**3 408**
Assembly		
Operation	`smmulr` (This thesis)	`smlaw{b, t}` (This thesis)
`Compress`$_1$	1 135	**959**
`Compress`$_4$	1 167	**991**
`Compress`$_5$	1 358	**1 152**
`Compress`$_{10}$	1 454	**1 247**
`Compress`$_{11}$	-	**1 335**

*Benchmark of this thesis.

For the compression functions `Compress`$_{\{1,4,5,10,11\}}$, instead of the straightforward assembly implementations of the Barrett-based compressions, this thesis exploits the DSP instructions `smmulr`, `smlawb`, and `smlawt` (cf. Section 9.3.2). In additional to the Barrett-based compressions, we can also pack the elements with the DSP instructions `pkhbt` and `pkhtb`. See Table 13.8 for a summary of the performance of the C implementations and the assembly-optimized implementations with `smmulr`, `smlawb`, and `smlawt`.

13.4.3 Scheme

For the Kyber implementations, this thesis integrates the new Barrett-based compressions into the implementations by [HZZ+22]. Table 13.9 summarizes the performance of [AHKS22], [HZZ+22], and the implementations of this thesis.

Table 13.9: Performance cycles of Kyber on Cortex-M4.

Parameter set	Work	**K**	**E**	**D**
`kyber512`	[AHKS22]*	389k	383k	422k
	[HZZ+22]*	**388k**	384k	420k
	This thesis	**388k**	**382k**	**418k**
`kyber768`	[AHKS22]*	**631k**	643k	695k
	[HZZ+22]*	**631k**	643k	691k
	This thesis	**631k**	**641k**	**688k**
`kyber1024`	[AHKS22]*	**998k**	1 005k	1 071k
	[HZZ+22]*	**998k**	1 005k	1 066k
	This thesis	**998k**	**1 003k**	**1 063k**

*Benchmark of this thesis.

13.5 Reviewing and Improving Armv8-A Neon Implementations

For the Armv8-A Neon implementations of Kyber, this thesis reviews the implementations by [ABD+20b, NG21, BHK+21, ABKK23], improves $\mathsf{NTT}/\mathsf{NTT}^{-1}$, and benchmarks the performance of [ABD+20b, NG21, BHK+21, ABKK23] and the improved implementations by this thesis.

13.5.1 Kyber `NTT`, $\mathtt{NTT}^{-1}$, MV, and IP

Kyber `NTT`/$\mathtt{NTT}^{-1}$. For the Kyber `NTT`/$\mathtt{NTT}^{-1}$ with Armv8-A Neon, [NG21] proposed Montgomery multiplication with the long multiplication instructions `s{mul, mla, mls}l{, 2}` and permutation instructions `uzp{1, 2}`, essentially following the non-vectorized 32-bit Montgomery multiplication on Cortex-M4 [ACC+20, GKS20]. [BHK+21] proposed the signed Barrett multiplication with the high multiplication instruction `sqrdmulh` (cf. Algorithm 9.19), completely removing the permutation instructions for twiddle-factor multiplications, proved a correspondence between Montgomery and Barrett multiplications, and improved the performance of `NTT`/$\mathtt{NTT}^{-1}$. [ABKK23] improved the instruction scheduling of `NTT` with an automatic tool and this thesis implements a more advanced instruction scheduling without any tooling. See Table 13.10 for a summary of the performance.

Table 13.10: Performance cycles of `NTT`s and $\mathtt{NTT}^{-1}$s in Kyber with Armv8-A Neon on Cortex-A72 and Firestorm.

Operation	Cortex-A72		Firestorm	
	`NTT`	$\mathtt{NTT}^{-1}$	`NTT`	$\mathtt{NTT}^{-1}$
[NG21]	1 473	1 661	413	428
[BHK+21]	1 200	1 368	263	262
[ABKK23]	**932**	-	-	-
This thesis	955	**1 130**	**225**	**228**

Matrix-vector multiplications and inner products. For the matrix-vector multiplications and inner products, [BHK+21] proposed asymmetric multiplication and applied the improved accumulation by [CHK+21] to the base multiplication. This thesis improves the instruction scheduling of `NTT`, $\mathtt{NTT}^{-1}$, and base multiplications. See Table 13.11 for a summary of the performance of matrix-vector multiplications and inner products by [NG21, BHK+21] and this thesis.

Table 13.11: Performance cycles of matrix-vector multiplications, inner products in encryptions, inner products in decryptions, and asymmetric multiplications of Kyber with Armv8-A Neon on Cortex-A72 and Firestorm.

Operation	Parameter set	[NG21]*	[BHK+21]*	This thesis
Cortex-A72				
MV	`kyber512`	10 700	6 849	**5 710**
	`kyber768`	19 300	11 077	**9 249**
	`kyber1024`	-	16 338	**13 754**
IP (Enc)	`kyber512`	7 100	2 000	**1 751**
	`kyber768`	9 900	2 242	**1 959**
	`kyber1024`	-	2 758	**2 335**
IP (Dec)	`kyber512`	7 100	4 844	**3 958**
	`kyber768`	9 900	6 518	**5 287**
	`kyber1024`	-	8 487	**6 793**
Asymmetric mul.	`kyber512`	-	963	**623**
	`kyber768`	-	1 166	**841**
	`kyber1024`	-	1 383	**1 033**
Firestorm				
MV	`kyber512`	2 809	1 431	**1 239**
	`kyber768`	4 910	2 291	**2 006**
	`kyber1024`	7 651	3 247	**2 876**
IP (Enc)	`kyber512`	1 858	412	**375**
	`kyber768`	2 545	461	**425**
	`kyber1024`	3 271	509	**475**
IP (Dec)	`kyber512`	1 858	926	**863**
	`kyber768`	2 545	1 232	**1 154**
	`kyber1024`	3 271	1 536	**1 446**
Asymmetric mul.	`kyber512`	-	176	**150**
	`kyber768`	-	224	**199**
	`kyber1024`	-	286	**263**

*Benchmark of this thesis.

13.5.2 `Compress`$_d$

As for `Compress`$_d$, this thesis implements C, intrinsic-optimized, assembly-optimized Barrett-based implementations. On Cortex-A72, the C and intrinsic-optimized implementations are compiled with `Ubuntu clang 15.0.7` and `Ubuntu GCC 12.3.0-1ubuntu1 23.04`. On Firestorm, the C and intrinsic-optimized implementations are compiled with `Apple clang 15.0.0`) and `Homebrew GCC 13.3.0`. Table 13.12 summarizes the performance of the C implementations. In Table 13.12, the Barrett-based compressions in C consistently outperform the C implementations in [ABD+20b] except for the clang-compiled $\texttt{Compress}_5$. Dumping the binaries of $\texttt{Compress}_5$ shows that some optimizations are introduced only when the input coefficients are non-negative numbers, which is the case for the C implementations by [ABD+20b] as the inputs are first mapped to the non-negative representation. It is unclear why the situations are swapped for the C implementations of $\texttt{Compress}_{10}$.

Table 13.12: Performance cycles of C implementations of $\texttt{Compress}_d$ in Kyber on Cortex-A72 and Firestorm.

Operation	GCC		Clang	
Cortex-A72				
	[ABD+20b]*	This thesis	[ABD+20b]*	This thesis
$\texttt{Compress}_1$	408	**314**	543	**454**
$\texttt{Compress}_4$	453	**231**	791	**435**
$\texttt{Compress}_5$	899	**642**	770	**649**
$\texttt{Compress}_{10}$	1 266	**481**	1 194	**385**
$\texttt{Compress}_{11}$	1 132	**900**	1 250	**753**
Firestorm				
	[ABD+20b]*	This thesis	[ABD+20b]*	This thesis
$\texttt{Compress}_1$	122	**86**	133	**119**
$\texttt{Compress}_4$	113	**45**	164	**107**
$\texttt{Compress}_5$	464	**280**	**176**	432
$\texttt{Compress}_{10}$	509	**188**	484	**85**
$\texttt{Compress}_{11}$	504	**376**	448	**353**

*Benchmark of this thesis.

As for the Neon-optimized implementations (cf. Section 9.3.3), this thesis implements intrinsic-optimized and assembly-optimized implementations. See Table 13.13 for a summary of the performance. From Table 13.13, the assembly-optimized implementations are either faster or comparable to the intrinsic-optimized implementations.

Table 13.13: Performance cycles of Armv8-A Neon implementations `Compress`$_d$ in Kyber on Cortex-A72 and Firestorm.

Operation	GCC		Clang	
Cortex-A72				
	Intrinsics	Assembly	Intrinsics	Assembly
`Compress`$_1$	189	**183**	**173**	179
`Compress`$_4$	170	**154**	179	**154**
`Compress`$_5$	822	**351**	525	**349**
`Compress`$_{10}$	981	**504**	813	**498**
`Compress`$_{11}$	826	**680**	830	**673**
Firestorm				
	Intrinsics	Assembly	Intrinsics	Assembly
`Compress`$_1$	**47**	57	**54**	57
`Compress`$_4$	**39**	47	**47**	**47**
`Compress`$_5$	160	**94**	138	**94**
`Compress`$_{10}$	210	**149**	237	**149**
`Compress`$_{11}$	**133**	146	210	**146**

13.5.3 Scheme

For the overall performance of Kyber, this thesis integrates the matrix-vector multiplication and inner products with improved instruction scheduling and the assembly-optimized Barrett-based compression functions. See Table 13.14 for a summary of the performance.

Table 13.14: Performance cycles of Kyber with Armv8-A Neon on Cortex-A72 and Firestorm.

Parameter set	Work	**K**	**E**	**D**
Cortex-A72				
`kyber512`	[BHK+21]*	64 209	67 831	78 352
	This thesis	**62 919**	**64 551**	**73 975**
`kyber768`	[BHK+21]*	102 660	115 980	130 524
	This thesis	**100 299**	**110 625**	**124 081**
`kyber1024`	[BHK+21]*	163 382	174 394	194 933
	This thesis	**160 421**	**167 420**	**186 949**
Firestorm				
`kyber512`	[BHK+21]*	17 819	19 660	26 101
	This thesis	**17 598**	**18 720**	**24 857**
`kyber768`	[BHK+21]*	29 241	32 061	41 491
	This thesis	**28 908**	**30 731**	**39 848**
`kyber1024`	[BHK+21]*	42 267	46 365	60 354
	This thesis	**41 888**	**43 711**	**57 306**

*Benchmark of this thesis.

13.6 Reviewing and Improving AVX2 Implementations

For the AVX2-optimized implementations of Kyber, this thesis reviews the implementations by [ABD+20b], improves the compressions, and benchmarks the implementations by [ABD+20b] and the improved implementations by this thesis.

13.6.1 Kyber NTT, NTT^{-1}, MV, and IP

For the Kyber NTT/NTT^{-1} with AVX2, the state-of-the-art approach is the Montgomery-based one by [Sei18] as shown in Algorithm 9.20, which was later integrated to [ABD+20b]. As for the matrix-vector multiplication and inner products, the state-of-the-art AVX2 implementations compute the products in the base multiplications one vector at a time. It is possible to further improve the performance with asymmetric multiplication [BHK+21] and the improved accumulation for the base multiplications [CHK+21]. See Table 13.15 for a summary of the performance of NTT and NTT^{-1}, and Table 13.16 for a summary of the performance of matrix-vector multiplications and inner products.

Table 13.15: Performance cycles of NTT, NTT^{-1}, and base multiplications of Kyber with AVX2 on Haswell.

Work	NTT	NTT^{-1}	Base mul.
[ABD+20b]*	281	280	136

*Benchmark of this thesis.

Table 13.16: Performance cycles of matrix-vector multiplications, inner products in the encryptions, and inner products in the decryptions of Kyber with AVX2 on Haswell.

Operation	Work	`kyber512`	`kyber768`	`kyber1024`
MV	[ABD+20b]*	1 720	3 080	4 747
IP (Enc)	[ABD+20b]*	582	736	890
IP (Dec)	[ABD+20b]*	1 136	1 593	2 016

*Benchmark of this thesis.

13.6.2 $\mathtt{Compress}_d$

For the AVX2-optimized compression functions $\mathtt{Compress}_{\{1,4,5,10,11\}}$, this thesis implements C and intrinsic-optimized implementations. For both versions, we compile with `gcc (Debian 12.2.0-14) 12.2.0` and `Debian clang version 14.0.6`. Table 13.17 summarizes the performance of the C implementations by [ABD+20b]. According to Table 13.17, the new Barrett-based C implementations are faster than or comparable to the C implementations by [ABD+20b] when compiled with GCC. As for the Clang-compiled implementations, the new Barrett-based C implementations consistently outperform the C implementations by [ABD+20b] except for $\mathtt{Compress}_1$ – for $\mathtt{Compress}_1$, they perform comparably.

Table 13.17: Performance cycles of C implementations of `Compress`$_d$ in Kyber on Haswell.

Operation	GCC		Clang	
	[ABD$^+$20b]*	This thesis	[ABD$^+$20b]*	This thesis
`Compress`$_1$	288	**285**	**397**	399
`Compress`$_4$	252	**216**	461	**398**
`Compress`$_5$	**491**	498	770	**649**
`Compress`$_{10}$	**1 311**	1 341	968	**694**
`Compress`$_{11}$	1 382	**1 381**	1 205	**959**

*Benchmark of this thesis.

As for the AVX2-optimized implementations (cf. Section 9.3.4), the new Barrett-based compressions outperform prior implementations by [ABD$^+$20b] except for `Compress`$_1$ – the prior approach computes the 1-bit output of `Compress`$_1$ with few comparisons to predefined constants, arithmetic shifts, subtractions, and logical or operations. The prior approach for `Compress`$_1$ remains the fastest one since no multiplications are involved.

Table 13.18: Performance cycles of AVX2 implementations of `Compress`$_d$ in Kyber on Haswell.

Operation	GCC		Clang	
	[ABD$^+$20b]*	This thesis	[ABD$^+$20b]*	This thesis
`Compress`$_1$	**42**	45	**34**	42
`Compress`$_4$	**41**	**41**	**40**	**40**
`Compress`$_5$	90	**75**	87	**74**
`Compress`$_{10}$	178	**152**	182	**139**
`Compress`$_{11}$	245	**215**	243	**210**

13.6.3 Scheme

For the overall performance of Kyber, this thesis integrates the new AVX2-optimized Barrett-based compressions for `Compress`$_{\{4,5,10,11\}}$. Table 13.19 summarizes the performance of the AVX2-optimized implementations.

Table 13.19: Performance cycles of Kyber with AVX2 on Haswell.

Parameter set	Work	**K**	**E**	**D**
`kyber512`	[ABD+20b]*	26 514	28 552	31 888
	This thesis	**26 389**	**28 477**	**31 800**
`kyber768`	[ABD+20b]*	45 922	45 545	49 966
	This thesis	**45 546**	**45 280**	**49 837**
`kyber1024`	[ABD+20b]*	**62 923**	**64 555**	71 311
	This thesis	62 944	64 692	**71 298**

*Benchmark of this thesis.

Chapter 14

NTRU

14.1 Specification

NTRU is a lattice-based KEM [CDH+20] in the 3rd round of the NIST PQC Standardization, and is based on the hardness of NTRU problem. Let q be an integer and n be a prime. Define $S_2 \coloneqq \mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$, $S_3 \coloneqq \mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$, $R_3 \coloneqq \mathbb{Z}_3[x]/\langle x^n - 1\rangle$, $S_q \coloneqq \mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$, $R_q \coloneqq \mathbb{Z}_q[x]/\langle x^n - 1\rangle$. In NTRU, n is chosen such that S_2 and S_3 are finite fields. See Table 14.1 for a summary of parameter sets.

Table 14.1: NTRU parameter sets.

Parameter set	q	n
`ntruhps2048509`	$2048 = 2^{11}$	509
`ntruhps2048677`	$2048 = 2^{11}$	677
`ntruhrss701`	$8192 = 2^{13}$	701
`ntruhps4096821`	$4096 = 2^{12}$	821
`ntruhps40961229`	$4096 = 2^{12}$	1229
`ntruhrss1373`	$16384 = 2^{14}$	1373

Key generation. During the key generation, we first generate two polynomials (f, g) from *seed*. We compute the inverse $f^{-1} \in S_2$, lift $f^{-1} \in S_2$ to $f_q \in S_q$, and compute the public polynomial $h = 3gf_q \in R_q$. We then compute the inverses $h_q = h^{-1} \in S_q$ and $f_p = f^{-1} \in S_3$. The public key is defined as `pkEncode`(h), and the secret key is defined as `skEncode` (f, f_p, h_q). See Algorithm 14.1 for an illustration.

Algorithm 14.1 NTRU Key Generation.

Input: *seed*.

Output:

$$\begin{cases} \text{Public key} & pk = \texttt{pkEncode}(h). \\ \text{Secret key} & sk = \texttt{skEncode}\left(f, f_p, h_q\right). \end{cases}$$

```
$(f, g) \in S_3^2 \leftarrow \texttt{Samplefg}(seed)$
$f_q \leftarrow f^{-1} \in S_q$
$h \leftarrow 3gf_q \in R_q$
$h_q \leftarrow h^{-1} \in S_q$
$f_p \leftarrow f^{-1} \in S_3$
$pk = \texttt{pkEncode}(h)$
$sk = \texttt{skEncode}\left(f, f_p, h_q\right)$
```

Encryption. Given a public polynomial h and a random polynomial r, we encrypt the message polynomial m by lifting it to m' and computing $rh + m' \in R_q$. The ciphertext is defined as `packq` $(rh + m' \in R_q)$. See Algorithm 14.2 for an illustration.

Algorithm 14.2 NTRU CPA Encryption.

Input:

$$\begin{array}{lll} \text{Public key} & pk & = \texttt{pkEncode}(h). \\ \text{Message} & m & \in \{0,1,2\}^{n-1} \times \{0\}. \\ \text{Randomness} & r & \in \{0,1,2\}^{n-1} \times \{0\}. \end{array}$$

Output: Ciphertext $ct \in \{0,1\}^{\left\lceil \frac{(n-1)\log_2 q}{8} \right\rceil}$.

```
$h \leftarrow \texttt{pkDecode}(pk)$
$m' \leftarrow \texttt{Lift}(m)$
$c \leftarrow (rh + m') \in R_q$
$ct \leftarrow \texttt{packq}(c)$
```

Decryption. For decrypting a ciphertext polynomial c with the secret polynomials f, f_p, h_q, we first test if $c \equiv 0 \pmod{q, \Phi_1(x)}$. If $c \not\equiv 0 \pmod{q, \Phi_1(x)}$, we return a failure and continue otherwise. We compute $a = cf \in R_q$ and $m = af_p \in S_3$, lift m to m', compute $r = (c - m')\, h_q \in S_q$, and return the message polynomial m and the random polynomial r. See Algorithm 14.3 for an illustration.

Algorithm 14.3 NTRU CPA Decryption.

```
Input:
    Ciphertext  $ct = \texttt{packq}(c)$.
    Secret key  $sk = \texttt{skEncode}(f, f_p, h_q)$.
Output: A pair of randomness $r$ and message $m$, or $\perp$.
c = unpackq(ct)
(f, f_p, h_q) = skDecode(sk)
if $c \not\equiv 0 \bmod (q, \Phi_1(x))$ then
    return $\perp$
$a \leftarrow cf \in R_q$
$m \leftarrow af_p \in S_3$
$m' \leftarrow \texttt{Lift}(m)$
$r \leftarrow (c - m')h_q \in S_q$
```

14.2 Optimization Guide for Polynomial Arithmetic

14.2.1 Polynomial Multiplication

The are two kinds of polynomial multiplications we need to implement in NTRU: (i) polynomial multiplication in $\mathbb{Z}_3[x]/\langle x^n - 1\rangle$, and (ii) polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$.

Multiplying polynomials in $\mathbb{Z}_3[x]/\langle x^n - 1\rangle$ with coefficient ring switching. For polynomial multiplication in $\mathbb{Z}_3[x]/\langle x^n - 1\rangle$, a straightforward approach is coefficient ring switching: We start with the signed representation $\mathbb{Z}_3 = \{-1, 0, 1\}$, and find that all the coefficients of the polynomial product in $\mathbb{Z}_3[x]/\langle x^n - 1\rangle$ have absolute values upper-bounded by n. Therefore, we choose a modulus $q' > 2n$, compute the product of two polynomials in $\mathbb{Z}_{q'}[x]/\langle x^n - 1\rangle$, and reduce the coefficients to $\mathbb{Z}_3$. Commonly, q' is chosen as an modulus defining a radix-2 Cooley–Tukey FFT as the dimension n is already large enough that NTTs outperform other asymptotically slower approaches such as Toom–Cook in practice.

Multiplying polynomials in $\mathbb{Z}_3[x]/\langle x^n - 1\rangle$ with integer multiplications. An alternative approach is to issue several coefficient ring arithmetic with integer multiplications and integer additions: We start with the unsigned representation $\mathbb{Z}_3 = \{0, 1, 2\}$ and store each coefficient as a byte. During the computation with 32-bit arithmetic, we load a 32-bit register containing a size-4 polynomial over $\mathbb{Z}_3$. With such an encoding, a 32-bit integer addition implements an addition of size-4 polynomials, and a 32-bit long integer multiplication computing a 64-bit long product implements a multiplication of size-4 polynomials. As long as there are no overflows in the byte-wise arithmetic, we can reduce each byte to $\{0, 1, 2\}$ and extract the desired coefficients. For reduction to $\{0, 1, 2\}$, we perform component-wise reduction with additions, and logical operations. See [Li21] for further details on the arithmetic.

Multiplying polynomials in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$. For polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$, similarly to Saber, we also have two lines of approaches. (i) If we compute entirely over $\mathbb{Z}_q$, we have to resort to Toom–Cook, Toeplitz-TC, Schönhage, and Nussbaumer as $\mathbb{Z}_q$ lacks roots of unity defining high-dimensional NTTs. (ii) We can also exploit the smallness of the secret polynomial, and choose a new modulus admitting efficient high-dimensional NTTs [CHK+21].

14.2.2 Polynomial Inversion

Polynomial inversion in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$ with extended Euclidean algorithm. For computing an inverse of an element in a ring, a straightforward approach is extended Euclidean algorithm. [BY19] proposed an asymptotically faster approach jumpdivstep compared to the constant-time divstep approach. The jumpdivstep approach is not a contribution of this thesis, and we refer to [BY19, JWYC24] for further details on vectorized implementations.

Polynomial inversions in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$ with bitslicing. For polynomial inversions in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$, a straightforward optimization is to exploit the inherent parallelism of the registers with bitslicing – for an arithmetic, we spell out the circuit mapping the inputs to the outputs and implement the circuit at the software layer. For example, in $\mathbb{Z}_2$, an addition translates into an exclusive-or and a multiplication translates into an and. For bitsliced $\mathbb{Z}_2$ arithmetic with `b`-bit registers, we pack `b` elements drawn from $\mathbb{Z}_2$ and compute `b` additions in $\mathbb{Z}_2$ with a single exclusive-or defined on the `b`-bit registers. Similarly, we compute `b` multiplications in $\mathbb{Z}_2$ with a single and defined on the `b`-bit registers. For bitslicing arithmetic in $\mathbb{Z}_3 = \{-1, 0, 1\}$, we follow the circuits from [BBC+20]: we split an element into a low bit and a high bit, and pack the low bits together and the high bits together. See Listings 14.1 and 14.2 for the C implementations with 32-bit variables. The implementation clearly scales with the growth of the bitsize of the variables. For example, for 64-bit platforms with 64-bit registers, we can implement the computation entirely with `int64_t` or `uint64_t`. We can also implement the bitsliced arithmetic with SIMD registers: in Neon, we implement with the 128-bit `q` registers, and in AVX2, we implement with the 256-bit `ymm` registers.

Listing 14.1: C implementation of bitsliced addition of elements in $\mathbb{Z}_3 = \{-1, 0, 1\}$ based on [BBC+20].

```
void bitsliced_add_signedZ3(
        int32_t *clo, int32_t *chi,
        const int32_t alo, const int32_t ahi,
        const int32_t blo, int32_t bhi){

    int32_t _clo, _chi, t;

    t = alo ^ bhi;
    _clo = (alo ^ blo) | (t ^ ahi);
    _chi = t & (ahi ^ blo);

    *clo = _clo;
    *chi = _chi;

}
```

Listing 14.2: C implementation of bitsliced multiplication of elements in $\mathbb{Z}_3 = \{-1, 0, 1\}$ based on [BBC+20].

```
void bitsliced_mul_signedZ3(
        int32_t *clo, int32_t *chi,
        const int32_t alo, const int32_t ahi,
        const int32_t blo, int32_t bhi){

    int32_t _clo, _chi;

    _clo = alo & blo;
    _chi = (ahi ^ bhi) & (_clo);

    *clo = _clo;
    *chi = _chi;
```

```
}
```

Polynomial inversion with Fermat's little theorem. Since $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$ are finite fields, it is natural to compute the inverses with Fermat's little theorem by raising an element to its (2^n-2)th power in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and (3^n-2)th power in $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$. For simplicity, we explain the exponentiation of an element in $\mathbb{Z}_p[x]/\langle\Phi_n(x)\rangle$ for coprime p and n. Instead of computing modulo $\Phi_n(x)$, we exponentiate modulo x^n-1 and reduce modulo $\Phi_n(x)$ at the end of whole exponentiation. For efficient exponentiation, we exploit the Frobenius homomorphism defined on $\mathbb{Z}_p[x]/\langle x^n-1\rangle$.

Theorem 12 (Frobenius homomorphism on $\mathbb{Z}_p[x]/\langle x^n-1\rangle$ as a permutation)**.** Let p be a prime, $n\perp p$ be a positive integer, and $\boldsymbol{a}$ be a polynomial in $\mathbb{Z}_p[x]/\langle x^n-1\rangle$. The Frobenius homomorphism $F=\boldsymbol{a}\mapsto\boldsymbol{a}^p$ permutes the coefficients of $\boldsymbol{a}$.

Proof. For $\boldsymbol{a}=\sum_{i=0}^{n-1}a_ix^i$, we have $\boldsymbol{a}^p=\sum_{i=0}^{n-1}a_ix^{pi}=\sum_{i=0}^{n-1}a_{ip^{-1}\bmod n}x^i$. □

For the target exponent p^n-2, we write it as a chain of additions and p^k-multiplying. Whenever an addition is encountered, we perform a polynomial multiplication in $\mathbb{Z}_p[x]/\langle x^n-1\rangle$, and whenever a p^k-multiplying is encountered, we simply permute the coefficients implementing F^k.

Polynomial inversion over $\mathbb{Z}_q$ with Hensel's lifting. For polynomial inversion in $\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$ for q a power of two, we compute the inverse in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and lift it to $\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$ with Hensel's lemma. Hensel's lemma says that a root of a univariate polynomial modulo a prime number p can be lifted to a unique root of the same univariate polynomial modulo a power of p. For an element $a\in\mathbb{Z}_q$, suppose we find an element $b\in\mathbb{Z}_2$ satisfying $ab\equiv 1 \pmod 2$. Obviously, b is a root of the polynomial $az-1$ in $\mathbb{Z}_2[z]$, and can be lifted to a root of $az-1$ over $\mathbb{Z}_q$. We recall the following for lifting an inverse constructively with Hensel's lemma, and name the approach as Hensel's lifting.

Theorem 13 (Hensel's lifting for inversion)**.** Let k_1 and k_2 be positive integers. For an element $a\in\mathbb{Z}_{p^{k_1+k_2}}$, and elements b_1 and b_2 satisfying $b_1\equiv a^{-1} \pmod{p^{k_1}}$ and $b_2\equiv a^{-1} \pmod{p^{k_2}}$, we have $b_1+b_2-ab_1b_2\equiv a^{-1} \pmod{p^{k_1+k_2}}$.

Proof. By assumption, there are integers h_1 and h_2 satisfying $ab_1=1+h_1p^{k_1}$ and $ab_2=1+h_2p^{k_2}$ in $\mathbb{Z}$. This implies $a\,(b_1+b_2-ab_1b_2)=1+h_1h_2p^{k_1+k_2}\in\mathbb{Z}$ so $b_1+b_2-ab_1b_2\equiv a^{-1} \pmod{p^{k_1+k_2}}$. □

Batch polynomial inversion with Montgomery's inversion. In the key generation, we have to compute the inverses of two polynomials in S_q. We recall below Montgomery's inversion computing a batch of inverses with a single inversion and several multiplications.

Definition 35 (Montgomery's inversion for batch inversion)**.** For a ring R and m invertible elements $a_0, \dots, a_{m-1} \in R$, Montgomery's inversion computes the inverses $a_0^{-1}, \dots, a_{m-1}^{-1} \in R$ with $2m - 2$ multiplications by elements drawn from $\{a_0, \dots, a_{m-1}\}$, $m - 1$ multiplications, and one inversion in R. The details are outlined as follows:

1. Compute the products in this order

$$a_0, a_0 a_1, \dots, \prod_{i=0}^{m-1} a_i$$

   with multiplications.

2. Invert the product $\prod_{i=0}^{m-1} a_i$.

3. Compute the inverses of products in this order

$$\left(\prod_{i=0,\dots,n-1} a_i \right)^{-1}, \left(\prod_{i=0,\dots,n-2} a_i \right)^{-1}, \dots, a_0^{-1}$$

   with multiplications.

4. Compute the inverse $a_j^{-1} = \left(\prod_{i=0,\dots,j} a_i \right)^{-1} \left(\prod_{i=0,\dots,j-1} a_i \right)$ for all $j = 1, \dots, m-1$ with multiplications.

Further, while computing the inverses of m polynomials $\boldsymbol{a}_0, \dots, \boldsymbol{a}_{m-1}$ with Montgomery's inversion, we can apply coefficient ring switching while multiplying by $\boldsymbol{a}_0, \dots, \boldsymbol{a}_{m-1}$ if the coefficients of $\boldsymbol{a}_0, \dots, \boldsymbol{a}_{m-1}$ are small integers.

14.3 Reviewing Cortex-M4 Implementations

For the Cortex-M4 implementations of NTRU, this thesis reviews the implementations [KRS19, CHK+21, IKPC22, AHY22], and reports the performance of [IKPC22, AHY22], which incorporated the improvement of fast constant-time GCD by [Li21].

14.3.1 Polynomial multiplication

For the polynomial multiplications in NTRU on Cortex-M4, there are several works: Toom–Cook [KRS19], Toeplitz-TC [IKPC22], and 32-bit NTT [CHK+21, AHY22]. [KRS19] computed the product with Toom–Cook over $\mathbb{Z}_{2^k}$ and implemented the 16-bit arithmetic with the DSP instructions. [CHK+21] adapted the 32-bit NTT approach for NTRU Prime by [ACC+20] to NTRU, and outperformed the Toom–Cook over $\mathbb{Z}_{2^k}$ by [KRS19]. [IKPC22] adapted the Toeplitz-TC approach for Saber by [IKPC20] to NTRU, and outperformed the 32-bit NTT approach by [CHK+21]. Finally, [AHY22] improved the transformation choices for the 32-bit NTT, and outperformed the Toeplitz-TC approach by [IKPC22].

Toom–Cook vs Toeplitz-TC. The Toeplitz-TC approach by [IKPC22] outperformed the Toom–Cook approach by [KRS19] since we save memory operations for the polynomial reduction.

32-bit NTTs. We take the 32-bit NTTs approaches by [CHK+21, AHY22] for `ntruhps2048677` as examples. While [CHK+21] computed over $x^{1536} - 1$, [AHY22] computed over a smaller polynomial modulus $x^{1440} - 1$. Since the input polynomials are zero-padded to the full size ones, we can skip the arithmetic defined on these entries. [CHK+21] applied the radix-2 Cooley–Tukey FFT and some of the additions and subtractions are omitted in the beginning. [AHY22] computed a size-288 DFT with Good–Thomas and Cooley–Tukey FFTs. They showed that one should compute the radix-3 butterflies in the beginning as one can save more from the zero-entries compared to computing radix-2 butterflies in the beginning.

Toeplitz-TC vs 32-bit NTT. As for the comparison between Toeplitz-TC and 32-bit NTT approaches, 32-bit NTT was slightly faster after [AHY22]'s improvement. See Table 14.2 for a summary of the performance of polynomial multiplications in `ntruhps2048677` and `ntruhrss701`.

Table 14.2: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ of `ntruhps2048677` and `ntruhrss701` on Cortex-M4.

Parameter set	[IKPC22]	[AHY22]	[AHY22]
	Size-$\{677, 701\}$	Size-1536	Size-1440
`ntruhps2048677`	144 825	148 438	**141 120**
`ntruhrss701`	145 029	148 838	**141 250**

14.3.2 Scheme

For the overall performance of NTRU, this thesis benchmarks the Toeplitz-TC approach by [IKPC22] and the 32-bit NTT approach by [AHY22] of `ntruhps2048677` and `ntruhrss701`. See Table 14.3 for a summary of the performance.

Table 14.3: Performance cycles of `ntruhps2048677` and `ntruhrss701` on Cortex-M4.

Parameter set	Work	**K**	**E**	**D**
`ntruhps2048677`	[AHY22]* (Size-1536)	3 970k	561k	710k
	[IKPC22]*	3 958k	557k	702k
	[AHY22]* (Size-1440)	**3 947k**	**554k**	**695k**
`ntruhrss701`	[AHY22]* (Size-1536)	3 861k	378k	768k
	[IKPC22]*	3 849k	374k	759k
	[AHY22]* (Size-1440)	**3 837k**	**371k**	**752k**

*Benchmark of this thesis.

14.4 Reviewing and Improving Armv8-A Neon Implementations

For the Armv8-A Neon implementations of NTRU, this thesis focuses on the parameter sets `ntruhps2048677` and `ntruhrss701`. We compare the implementations by [NG21, CCHY24], and the improved implementations by this thesis.

14.4.1 Polynomial Multiplication

For the polynomial multiplication in NTRU with Armv8-A Neon, there are several approaches: the Toom–Cook by [NG21, CCHY24] and the Toeplitz-TC by [CCHY24].

Table 14.4: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ of `ntruhps2048677` and `ntruhrss701` with Armv8-A Neon on Cortex-A72 and Firestorm.

Implementation	`ntruhps2048677`	`ntruhrss701`
Cortex-A72		
[NG21]* (TC)	55 427	57 040
[CCHY24]* (TC)	37 180	-
[CCHY24]* (Toeplitz-TC)	**26 907**	**31 151**
Firestorm		
[NG21]* (TC)	10 402	11 884
[CCHY24]* (TC)	7 187	-
[CCHY24]* (Toeplitz-TC)	**5 226**	**6 971**

*Benchmark of this thesis.

Toom–Cook: Toom-4 vs Toom-5. We first compare the Toom–Cook by [NG21, CCHY24]. In Armv8-A Neon, since each SIMD register holds eight 16-bit coefficients, the goal is to design efficient computations over chunks of 8-tuples. In both [NG21] and [CCHY24], they computed the products of size-720 polynomials. [NG21] decomposed a size-720 polynomial with two Toom-3's, one Toom-4, and two Karatsuba's, resulting in size-15 polynomial multiplications. Since 15 is very close to 16, they essentially zero-padded the small-dimensional polynomials to size-16 ones. [CCHY24] decomposed more aggressively with one Toom-5, two Toom-3's, and one Karatsuba, resulting in size-8 polynomial multiplications, avoiding the zero-padding. Along with some memory operations, the Toom–Cook by [CCHY24] significantly outperformed the Toom–Cook by [NG21].

Toeplitz-TC vs Toom-Cook. As for the comparisons between Toeplitz-TC and Toom–Cook, since Toeplitz-TC can be nicely mapped to vector-by-scalar multiplication instructions (cf. Section 7.3), [CCHY24]'s Toeplitz-TC outperformed their own Toom–Cook approach. See Table 14.4 for a summary of the performance of the Toom–Cook by [NG21], and the Toom–Cook and Toeplitz-TC by [CCHY24].

14.4.2 Polynomial Inversion

Table 14.5: Performance cycles of polynomial inversions in S_2, S_3, and S_q of `ntruhps2048677` and `ntruhrss701` with Armv8-A Neon on Cortex-A72 and Firestorm.

Operation	Work	`ntruhps2048677`	`ntruhrss701`
Cortex-A72			
S_2	[CDH+20]*	3 059 958	668 396
	[CCHY24]*	**135 439**	**59 257**
S_3	[CDH+20]*	4 429 570	773 756
	[CCHY24]*	486 319	151 770
	This thesis	**311 895**	**100 446**
S_q	[NG21]*	3 499 592	757 932
	[CCHY24]* (TC)	434 655	117 120
	[CCHY24]* (Toeplitz-TC)	**349 607**	**102 448**
Firestorm			
S_2	[CDH+20]*	3 299 673	745 970
	[CCHY24]*	**140 295**	**42 422**
S_3	[CDH+20]*	4 703 158	828 169
	[CCHY24]*	504 033	186 218
	This thesis	**322 939**	**104 000**
S_q	[NG21]*	3 765 628	842 248
	[CCHY24]* (Toeplitz-TC)	**392 053**	**99 658**

*Benchmark of this thesis.

$\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$. For the polynomial inversion in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$, the bitsliced implementation (cf. Section 14.2) by [CCHY24] significantly improved the performance.

$\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$. For the polynomial inversion in $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$, the bitsliced implementation by [CCHY24] also significantly improved the performance.

This thesis further improves the performance by replacing the circuits with the circuits by [BBC+20].

$\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$. For the polynomial inversion in $\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$, there are two big computations: the inversion in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ and the Hensel's lifting lifting an inverse in $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ to $\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$. In [CCHY24], they improved the performance of $\mathbb{Z}_2[x]/\langle\Phi_n(x)\rangle$ with bitslicing and the performance of Hensel's lifting with the improved polynomial multiplication in $\mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle$. See Table 14.5 for a summary of the performance.

14.4.3 Scheme

For the overall performance of NTRU, this thesis integrates the new bitsliced polynomial inversion in $\mathbb{Z}_3[x]/\langle\Phi_n(x)\rangle$ into [CCHY24], and benchmarks the implementations of `ntruhps2048677` and `ntruhrss701` by [NG21, CCHY24]. See Tables 14.6 and 14.7 for a summary of the performance.

Table 14.6: Performance cycles of `ntruhps2048677` with Armv8-A Neon on Cortex-A72 and Firestorm.

Work	**K**	**E**	**D**
Cortex-A72			
[NG21]* (TC)	8 308 232	110 511	207 834
[CCHY24]* (TC)	1 146 348	90 976	149 769
[CCHY24]* (Toeplitz-TC)	1 022 006	**81 976**	**122 050**
This thesis	**840 025**	-	-
Firestorm			
[NG21]* (TC)	2 195 500	30 499	50 043
[CCHY24]* (TC)	331 008	25 128	36 102
[CCHY24]* (Toeplitz-TC)	312 422	**23 303**	**30 510**
This thesis	**233 353**	-	-

*Benchmark of this thesis.

Table 14.7: Performance cycles of `ntruhrss701` with Armv8-A Neon on Cortex-A72 and Firestorm.

Work	**K**	**E**	**D**
Cortex-A72			
[NG21]* (TC)	8 837 561	87 291	222 199
[CCHY24]* (Toeplitz-TC)	1 076 673	**59 692**	**142 807**
This thesis	**895 154**	-	-
Firestorm			
[NG21]* (TC)	2 403 884	22 800	55 862
[CCHY24]* (Toeplitz-TC)	330 092	**16 588**	**37 442**
This thesis	**244 732**	-	-

*Benchmark of this thesis.

14.5 Reviewing AVX2 Implementations

For the AVX2-optimized implementations, this thesis reports the performance cycles of [ZCH+19, CHK+21] on Skylake.

14.5.1 Polynomial Multiplication

Table 14.8: Performance cycles of polynomial multiplications for NTRU with AVX2 on Skylake [CHK+21].

Parameter set	[ZCH+19]*	[CHK+21]
`ntruhps2048509`	**6 643**	8 540
`ntruhps2048677`	11 103	**10 373**
`ntruhrss701`	11 242	**10 373**
`ntruhps4096821`	15 507	**13 247**

*Reported by [CHK+21].

For the AVX2-optimized polynomial multiplications, [ZCH+19] implemented a 16-bit Toom–Cook operating over $\mathbb{Z}_{2^k}$, and [CHK+21] proposed the AVX2-

optimized 16-bit NTT implementations. Since the dimension of the input polynomials are somewhat large, the NTT approach is faster than the Toom–Cook approach for the parameter sets `ntruhps2048677`, `ntruhrss701`, and `ntruhps4096821`. See Table 14.8 for a summary of the performance.

14.5.2 Scheme

For the overall performance of NTRU with AVX2 on Skylake, this thesis reports the performance cycles reported in [CHK+21]. For the parameter sets `ntruhps2048677`, `ntruhrss701`, and `ntruhps4096821`, the 16-bit NTT approach is faster than the Toom–Cook approach. See Table 14.9 for a summary of the performance.

Table 14.9: Performance cycles of NTRU with AVX2 on Skylake [CHK+21].

Parameter set	Work	**K**	**E**	**D**
`ntruhps2048509`	[ZCH+19]*	**208 653**	**71 018**	**38 950**
	[CHK+21]	218 887	73 176	42 953
`ntruhps2048677`	[ZCH+19]*	332 906	96 293	59 169
	[CHK+21]	**333 278**	**95 953**	**58 406**
`ntruhrss701`	[ZCH+19]*	299 066	56 616	62 503
	[CHK+21]	**298 505**	**56 084**	**61 199**
`ntruhps4096821`	[ZCH+19]*	458 614	114 986	74 182
	[CHK+21]	**451 664**	**113 935**	**70 917**

*Reported by [CHK+21].

Chapter 15

NTRU Prime

15.1 Specification

NTRU Prime is a set of lattice-based KEMs [BBC+20] in the 3rd round of the NIST PQC Standardization. There are two KEMs in NTRU Prime: (i) Streamlined NTRU Prime based on the hardness of the NTRU problem and (ii) NTRU LPrime based on the hardness of R-LWR problem. Similar to NTRU, NTRU Prime involves two polynomial rings. However, in NTRU Prime, the polynomial ring over $\mathbb{Z}_q$ is chosen as a finite field with large Galois group and inert modulus. Let p, q be prime numbers such that $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle \cong \mathbb{F}_{q^p}$ and define $R_q := \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and $R_3 := \mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$. Notice that R_3 is not a field in NTRU Prime.

15.1.1 Streamlined NTRU Prime

Table 15.1: Streamlined NTRU Prime parameter sets.

Parameter set	p	q	w
`sntrup653`	653	4621	288
`sntrup761`	761	4591	286
`sntrup857`	857	5167	322
`sntrup953`	953	6343	396
`sntrup1013`	1013	7177	448
`sntrup1277`	1277	7879	492

Streamlined NTRU Prime is based on the NTRU problem. See Table 15.1 for a summary of parameter sets.

Key generation. In the key generation, we generate a polynomial $g \in \mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ until g is invertible in $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$. In practice, the inversion test is integrated into polynomial inversion in $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$. Then, we generate a polynomial $f \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, compute its inverse $f^{-1} \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, and compute $h = gf^{-1} \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$. The public key is defined as $\texttt{pkEncode}(h)$, and the secret key is defined as $\texttt{skEncode}\left(f, g^{-1}\right)$. See Algorithm 15.1 for an illustration.

Algorithm 15.1 Streamlined NTRU Prime key generation.

Output:

$$\begin{array}{lll} \text{Public key} & pk & = \texttt{pkEncode}(h). \\ \text{Secret key} & sk & = \texttt{skEncode}\left(f, g^{-1}\right). \end{array}$$

```
$g \leftarrow 0$
while $g^{-1} \in R_3 = \perp$ do
    $g \in R_3 \leftarrow \texttt{SmallRandom}()$
$f \in R_q \leftarrow \texttt{ShortRandom}()$
$f' \leftarrow f^{-1} \in R_q$
$h \leftarrow f'g \in R_q$
$pk \leftarrow \texttt{pkEncode}(h)$
$sk \leftarrow \texttt{skEncode}\left(f, g^{-1}\right)$
```

Algorithm 15.2 Streamlined NTRU Prime encryption.

Input:

$$\begin{array}{lll} \text{Public key} & pk & = \texttt{pkEncode}(h). \\ \text{Message} & r & \in \operatorname{Img}(\texttt{ShortRandom}). \end{array}$$

Output Ciphertext $ct = \texttt{RoundedEncode}(c)$.

```
$h \leftarrow \texttt{pkDecode}(pk)$
$c_q \leftarrow hr \in R_q$
$c \leftarrow \texttt{Round}(c_q)$
$ct \leftarrow \texttt{RoundedEncode}(c)$
```

Encryption. For encrypting a message polynomial r with the public polynomial h, we compute $hr \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and round it. The ciphertext is defined as $(\texttt{RoundedEncode} \circ \texttt{Round})(hr)$. See Algorithm 15.2 for an illustration.

Decryption. For decrypting the ciphertext polynomial c with secret polynomials f, g^{-1}, we compute $e_q = 3cf \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, map it to $e = e_q \bmod 3$, and compute $e_g = eg^{-1} \in \mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$. If the non-zero entries of e_g is exactly w, then we return it as the message polynomial and reject it otherwise.

Algorithm 15.3 Streamlined NTRU Prime decryption.

```
Input:
    Ciphertext  $ct$  $= \texttt{RoundedEncode}(c)$.
    Secret key  $sk$  $= \texttt{skEncode}\left(f, g^{-1}\right)$.
Output Message $r \in \mathrm{Img}\,(\texttt{ShortRandom})$ or $\perp$.
$\left(f, g^{-1}\right) \leftarrow \texttt{skDecode}(sk)$
$c \leftarrow \texttt{RoundedDecode}(ct)$
$e_q \leftarrow 3cf \in R_q$
$e \leftarrow e_q \in R_3$
$e_g \leftarrow eg^{-1} \in R_3$
if $\texttt{Weightw}\,(e_g) = \top$ then
    $r \leftarrow e_g$
else
    return $\perp$.
```

15.1.2 NTRU LPrime

NTRU LPrime is based on the R-LWR problem. See Table 15.2 for a summary of parameter sets.

Table 15.2: NTRU LPrime parameter sets.

Parameter set	p	q	w
`ntrulpr653`	653	4621	252
`ntrulpr761`	761	4591	250
`ntrulpr857`	857	5167	281
`ntrulpr953`	953	6343	345
`ntrulpr1013`	1013	7177	392
`ntrulpr1277`	1277	7879	429

Key generation. For the key generation, we expand the seed S into a polynomial $G \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, generate a polynomial $a \in \mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$, compute $aG \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, and round $A = \texttt{Round}\,(aG)$. The public key is defined as $(S, \texttt{RoundedEncode}(A))$, and the secret key is defined as $\texttt{skEncode}(a)$. See Algorithm 15.4 for an illustration.

Algorithm 15.4 NTRU LPrime key generation.

```
Input: Randomness $S$.
Output:
        Public key  $pk = (S, \texttt{RoundedEncode}(A))$.
        Secret key  $sk = \texttt{skEncode}(a)$.
$G \in R_q \leftarrow \texttt{ExpandG}(S)$
$a \in R_3 \leftarrow \texttt{ShortRandom}()$
$A' \leftarrow aG \in R_q$
$A \leftarrow \texttt{Round}\,(A')$
$pk \leftarrow (S, \texttt{RoundedEncode}(A))$
$sk \leftarrow \texttt{skEncode}(a)$
```

Algorithm 15.5 NTRU LPrime encryption.

```
Input:
        Public key  $pk = (S, \texttt{RoundedEncode}(A)$.
        Message     $r \in \{0,1\}^{256}$.
Output: Ciphertext $ct = (\texttt{RoundedEncode}(B), \texttt{TopEncode}(T))$.
$G \in R_q \leftarrow \texttt{ExpandG}(S)$
$b \in R_3 \leftarrow \texttt{HashShort}(r)$
$B' \leftarrow bG \in R_q$
$B \leftarrow \texttt{Round}\,(B')$
$A \leftarrow \texttt{RoundedDecode}\,(\texttt{RoundedEncode}(A))$
$A' \leftarrow bA \in R_q$
$T \in \{0,\ldots,15\}^{256} \leftarrow \texttt{Top}\left(A' + \frac{r(q-1)}{2}\right)$
$ct \leftarrow (\texttt{RoundedEncode}(B), \texttt{TopEncode}(T))$
```

Encryption. For encrypting a message polynomial r with public seed S, we expand S into $G \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and hash r to a polynomial $b \in \mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$. We then compute the product $bG \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ and round it to B. As for the public polynomial A, we also compute its product with b in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, and perform component-wise addition with $\frac{r(q-1)}{2}$ followed by top-bit extractions. The ciphertext is defined as $(\texttt{RoundedEncode}(B), \texttt{TopEncode}(T))$. See Algorithm 15.5 for an illustration. We refer to [BBC$^+$20] for the definition of the `Top` function.

Decryption. For decrypting ciphertext polynomials B, T with the secret polynomial a, we compute the product $aB \in \mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, subtract the product from $\texttt{Right}(T)$, component-wisely add with $4w + 1$, and component-wisely extract the signs of the resulting polynomial. See Algorithm 15.6 for an illustration. We refer to [BBC$^+$20] for the definition of the `Right` function.

Algorithm 15.6 NTRU LPrime decryption.

```
Input:
        Ciphertext  $ct = (\texttt{RoundedEncode}(B), \texttt{TopEncode}(T))$.
        Secret key  $sk = \texttt{skEncode}(a)$.
Output Message $r \in \{0,1\}^{256}$.
1: $a = \texttt{skDecode}(sk)$
2: $B \leftarrow \texttt{RoundedDecode}(\texttt{RoundedEncode}(B))$
3: $T \leftarrow \texttt{TopDecode}(\texttt{TopEncode}(T))$
4: $B' \leftarrow aB \in R_q$
5: $r \in \{0,1\}^{256} \leftarrow \texttt{sign}\left(\left(\texttt{Right}(T) - B' + (4w+1)\sum_{i=0}^{p-1} x^i\right) \in R_q\right)$
```

15.2 Optimization Guide for Polynomial Arithmetic

15.2.1 Polynomial Multiplication

For polynomial multiplication in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$, there are also two lines of strategies: (i) we compute entirely over $\mathbb{Z}_q$ and (ii) we apply coefficient ring switching.

Toom–Cook, Toeplitz-TC, Schönhage, and Nussbaumer over $\mathbb{Z}_q$. If we compute entirely over $\mathbb{Z}_q$, we can combine Toom–Cook, Toeplitz-TC, Schönhage, and Nussbaumer in various ways. Notice that as all these approaches result in increasingly larger number of coefficients. Since coefficient ring multiplications in NTRU Prime are much slower than the ones in NTRU, this line of approaches, although generically applied to all the parameter sets, are not very fast.

Coefficient ring switching. An alternative approach is coefficient ring switching [BBC+20, ACC+20]. If one of the operands has coefficients drawn from $\{-1, 0, 1\}$, we can choose a much smaller modulus as the new modulus, and compute smooth-dimensional Cooley–Tukey FFTs accordingly. Different from NTRU, the coefficients grow as large as $2p - 1$ in absolute values if we want to compute the products in $\mathbb{Z}[x]/\langle x^p - x - 1\rangle$.

Exploiting the special structure of $\mathbb{Z}_q$: a case study with $q = 4591$. While multiplying polynomials over $\mathbb{Z}_q$, we can also exploit the special structure of the modulus q. For the modulus q in NTRU Prime, $q - 1$ is not divisible by 4 and we do not have high-dimensional radix-2 Cooley–Tukey FFT. Nevertheless, we can still employ non-Cooley–Tukey FFTs when some conditionals are met. We demonstrate with the case $q = 4591$. Observe that $q + 1 = 2^4 \cdot 7 \cdot 41$. This implies $x^{2^k} + 1$ factors into irreducible polynomials of the form $x^2 + \gamma x + 1$ if $k < 4$ and $x^{2^{k-3}} + \gamma x^{2^{k-4}} - 1$ if $k \geq 4$. Therefore, we can multiply polynomials in $\mathbb{Z}_q[x] \Big/ \left\langle x^{2^k} + 1 \right\rangle$ efficiently with Bruun's FFT when k is not large. Next, $17 | (q - 1)$ implies Radix-17 FFT implementing $\mathbb{Z}_q[x]/\langle x^{17m} - 1\rangle \cong \prod_{i=0}^{16} \mathbb{Z}_q[x]/\langle x^m - \omega_{17}^i\rangle$ and also its truncation $\mathbb{Z}_q[x]/\langle \Phi_{17}(x^m)\rangle \cong \prod_{i=1}^{16} \mathbb{Z}_q[x]/\langle x^m - \omega_{17}^i\rangle$.

15.2.2 Polynomial Inversion

Polynomial inversion over $\mathbb{Z}_3$. For polynomial inversion over $\mathbb{Z}_3$, we can similarly compute with bitsliced extended GCD and the asymptotically faster constant-time extended GCD by [BY19]. Different from NTRU, Fermat's little theorem does not work as $\mathbb{Z}_3[x]/\langle x^p - x - 1\rangle$ is not a finite field.

Polynomial inversion over $\mathbb{Z}_q$. For polynomial inversion over $\mathbb{Z}_q$, the fastest approach is to apply the fast constant-time extended GCD by [BY19].

15.3 Reviewing Cortex-M4 Implementations

For the Cortex-M4 implementations of NTRU Prime on Cortex-M4, this thesis focuses on the parameter set `sntrup761`, reviews the implementations by [ACC+20, Che21, AHY22], and benchmarks the implementations by [Che21, AHY22].

15.3.1 Polynomial Multiplication

For the polynomial multiplications of NTRU Prime on Cortex-M4, we review the implementations targeting the parameter set `sntrup761` operating in the polynomial ring $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$. There are three lines of approaches in the literature: the Toom–Cook over $\mathbb{Z}_{4591}$ [ACC+20], the Rader's FFT over $\mathbb{Z}_{4591}$ [ACC+20, Che21], and the 32-bit NTT over $\mathbb{Z}_{q'}$ with coefficient ring switching [ACC+20, AHY22].

Toom–Cook vs NTT. Since Toom–Cook maps the input tuple to a tuple with a larger number of coefficients, the bottleneck of Toom–Cook is the small-dimensional polynomial multiplications over $\mathbb{Z}_{4591}$. Furthermore, multiplications in $\mathbb{Z}_{4591}$ amount to actual modular multiplications and the small-dimensional polynomial multiplications over $\mathbb{Z}_{4591}$ are not very fast. This renders slower implementations when compared to NTT approaches. In the NTT approaches, including Rader's FFT over $\mathbb{Z}_{4591}$ and 32-bit NTT over $\mathbb{Z}_{q'}$, the total number of coefficients remains the same.

Table 15.3: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^p - x - 1\rangle$ of `sntrup761` on Cortex-M4.

Parameter set	[Che21]**	[AHY22]*
	Size-1530	Size-1536
`sntrup761`	**142 615**	151 696

*Benchmark of this thesis.
**Benchmark of this thesis of implementations in https://github.com/mupq/pqm4/commit/9ff685e0ffbfdbafb745cb6ff56ca3f549173f12.

Rader's FFT over $\mathbb{Z}_{4591}$ vs 32-bit NTT over $\mathbb{Z}_{q'}$. As for the comparisons between Rader's FFT over $\mathbb{Z}_{4591}$ and 32-bit NTT over $\mathbb{Z}_{q'}$, Rader's FFT is slightly faster if we implement the coefficient ring arithmetic with the DSP extension in Armv7E-M. See Table 15.3 for a summary of the performance.

15.3.2 Scheme

For the overall performance of `sntrup761` on Cortex-M4, this thesis benchmarks the performance of the implementations by [Che21, AHY22]. See Table 15.4 for a summary of the performance.

Table 15.4: Performance cycles of `sntrup761` on Cortex-M4.

Parameter set	Work	**K**	**E**	**D**
`sntrup761`	[AHY22]* (size-1536)	8 019k	701k	561k
	[Che21]** (size-1530)	**8 014k**	**691k**	**536k**

*Benchmark of this thesis.
**Benchmark of this thesis of implementations in https://github.com/mupq/pqm4/commit/9ff685e0ffbfdbafb745cb6ff56ca3f549173f12.

15.4 Reviewing and Improving Armv8-A Neon Implementations

This section goes through detailed studies of the polynomial multiplications in $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$ for the parameter set `sntrup761` with Armv8-A Neon.

15.4.1 Polynomial Multiplication

For multiplying polynomials in $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$ with vector instructions with bit-size a power of two, such as Armv8-A Neon and AVX2, it is essential to work with polynomial rings of power-of-two dimensions (cf. Section 7.4). As $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$ has dimension close to 768, it is natural to choose a polynomial modulus with degree 1536. There are multiple options: $\left(x^{1024}+1\right)\left(x^{512}-1\right)$, $\left(x^{1024}+1\right)\left(x^{512}+1\right)$, $x^{1536}-1$, and $\Phi_{17}\left(x^{96}\right)$. For the former two options, essentially we design a fast transformation for $\mathbb{Z}_q[x]/\langle x^{2048} - 1\rangle$ and apply truncation as both polynomial moduli are factors of $x^{2048} - 1$. Since 2048 is a high-dimensional power of two, one can apply a combination of Toom–Cook, Toeplitz-TC, Schönhage, and Nussbaumer. This results in several small-dimensional polynomial multiplications over $\mathbb{Z}_q$, whose modular multiplications are not very fast. In the remainder of this section, we study the case $\mathbb{Z}_q = \mathbb{Z}_{4591}$ and go through three approaches gradually exploiting the structure of the coefficient ring for transformation. While stating the conditions, we explicitly spell out $q = 4591$ for the ease of validation.

15.4.1.1 Approach 1: Good–Thomas with Coefficient Ring Switching

An obvious approach is to compute the product over a new NTT-friendly modulus q'. Since one of the input polynomials has coefficients drawn from $\{-1, 0, 1\}$, it suffices to choose a 32-bit NTT-friendly modulus q'. [HLY24] chose a $\mathbb{Z}_{q'}$ containing a principal 384th root of unity and applied Good–Thomas and Cooley–Tukey FFTs. Since each SIMD registers in Armv8-A Neon contains four words, we do not need any permutation instructions.

15.4.1.2 Approach 2: Schönhage and Nussbaumer

The second approach is a transformation built upon Schönhage and Nussbaumer proposed by [BBCT22]. Starting from the ring $\mathbb{Z}_q[x]/\langle x^{2048}-1\rangle$, Schönhage's FFT computes the following:

$$\frac{\mathbb{Z}_q[x]}{\langle x^{2048}-1\rangle} \cong \frac{\mathbb{Z}_q[x]/\langle x^{32}-y\rangle\ [y]}{\langle y^{64}-1\rangle} \hookrightarrow \frac{\mathbb{Z}_q[x]/\langle x^{64}+1\rangle\ [y]}{\langle y^{64}-1\rangle} \cong \left(\frac{\mathbb{Z}_q[x]}{\langle x^{64}+1\rangle}\right)^{64}.$$

For polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^{64}+1\rangle$, we apply Nussbaumer as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^{64}+1\rangle} \cong \frac{\mathbb{Z}_q[x]/\langle z^{8}+1\rangle\ [x]}{\langle x^{8}-z\rangle} \hookrightarrow \frac{\mathbb{Z}_q[x]/\langle z^{8}+1\rangle\ [x]}{\langle x^{16}-1\rangle} \cong \left(\frac{\mathbb{Z}_q[x]}{\langle z^{8}+1\rangle}\right)^{16}.$$

In summary, $\mathbb{Z}_q[x]/\langle x^{2048}-1\rangle$ is transformed into $16 \cdot 64 = 1024$ copies of $\mathbb{Z}_q[z]/\langle x^8+1\rangle$. If we start with $\mathbb{Z}_q[x]/\langle (x^{1024}+1)\,(x^{512}-1)\rangle$, then it is transformed into $16 \cdot 48 = 768$ copies of $\mathbb{Z}_q[z]/\langle z^8+1\rangle$.

15.4.1.3 Approach 3: Schönhage, Good–Thomas, and Bruun

The third approach applies Schönhage's, Good–Thomas, and Bruun's FFTs, and results in half of the size-8 polynomial multiplications that are slight more expensive than computing in $\mathbb{Z}_q[x]/\langle x^8+1\rangle$ [HLY24]. The choices of polynomial ring and transformation are based on the following observations:

- As $3|(4591-1)$, we can define a radix-3 FFT. The challenging task is to combine the multiplicative radix-3 FFT with the symbolic radix-2 Schönhage.
- Although $4 \not| (4591-1)$ precludes high-dimensional radix-2 Cooley–Tukey FFT, $16|\,(4591+1)$ implies some medium-dimensional radix-2 Bruun's FFT (cf. Section 4.4).

Truncated Schönhage vs Good–Thomas and Schönhage in theory. We demonstrate how to combine a multiplicative radix-3 FFT with a symbolic radix-2 Schönhage. Instead of computing in $\mathbb{Z}_q[x]/\langle(x^{512}-1)(x^{1024}+1)\rangle$, we apply Schönhage's and Good–Thomas FFTs to $\mathbb{Z}_q[x]/\langle x^{1536}-1\rangle$. By definition, if $\boldsymbol{\omega}$ is a principal 2^kth root of unity and ω_3 is a principal 3rd root of unity, then $\omega_3\boldsymbol{\omega}$ is a principal $3\cdot 2^k$th root of unity. We introduce a principal 32nd root of unity $\boldsymbol{\omega}_{32}=x^2$ as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^{1536}-1\rangle}\cong\frac{\mathbb{Z}_q[x]/\langle x^{16}-y\rangle\,[y]}{\langle y^{96}-1\rangle}\hookrightarrow\frac{\mathbb{Z}_q[x]/\langle x^{32}+1\rangle\,[y]}{\langle y^{96}-1\rangle}.$$

Since $\omega_3\boldsymbol{\omega}_{32}$ is a principal 96th root of unity, we have

$$\frac{\mathbb{Z}_q[x]/\langle x^{32}+1\rangle\,[y]}{\langle y^{96}-1\rangle}\cong\prod_{i=0,\ldots,95}\frac{\mathbb{Z}_q[x]/\langle x^{32}+1\rangle\,[y]}{\langle y-\omega_3^i\boldsymbol{\omega}_{32}^i\rangle}.$$

However, one should not implement this isomorphism with mixed-radix Cooley–Tukey FFT. Observe that multiplication by $\boldsymbol{\omega}_{32}=x^2$ amounts to negating and permuting whereas multiplication by ω_3 amounts actual modular multiplication. Cooley–Tukey FFT requires one to multiply by $\omega_3^i\boldsymbol{\omega}_{32}^i$ which is unreasonably complicated to optimize when $i\perp 96$. We apply Good–Thomas FFT implementing

$$\frac{\mathbb{Z}_q[x]/\langle x^{32}+1\rangle\,[y]}{\langle y^{96}-1\rangle}\cong\frac{\mathbb{Z}_q[x]/\langle x^{32}+1\rangle\,[y]}{\langle y-uw,u^3-1,w^{32}-1\rangle}.$$

Obviously, we only need multiplications by powers of ω_3 and $\boldsymbol{\omega}_{32}$ and not $\omega_3\boldsymbol{\omega}_{32}$. See Table 15.5 for comparisons.

Table 15.5: Approaches for computing the size-1536 product of two polynomials drawn from $\mathbb{Z}_q[x]/\langle x^{761}-x-1\rangle$ with radix-2 Schönhage.

Approach	Domain	Image	Twiddle factors
Truncated [BBCT22]	$\frac{\mathbb{Z}_q[x]}{\langle(x^{1024}+1)(x^{512}-1)\rangle}$	$\left(\frac{\mathbb{Z}_q[x]}{\langle x^{64}+1\rangle}\right)^{48}$	x^{2i}
With Cooley–Tukey	$\frac{\mathbb{Z}_q[x]}{\langle x^{1536}-1\rangle}$	$\left(\frac{\mathbb{Z}_q[x]}{\langle x^{32}+1\rangle}\right)^{96}$	$\omega_3^i x^{2j}$
With Good–Thomas	$\frac{\mathbb{Z}_q[x]}{\langle x^{1536}-1\rangle}$	$\left(\frac{\mathbb{Z}_q[x]}{\langle x^{32}+1\rangle}\right)^{96}$	ω_3^i, x^{2j}

Good–Thomas and Schönhage in practice. We detail the implementations as follows.

- In practice, the replacement of $x^{16} \sim y$ by $x^{32} \sim -1$ is merged with the Good–Thomas permutation. We follow the vectorization-friendly Good–Thomas permutation by [AHY22, Section 3.3].

- Recall that one of the input polynomials has coefficients drawn from $\{-1, 0, 1\}$. We start with the 8-bit form of this polynomial and perform five layers of radix-2 butterflies without any modular reductions. The initial three layers of radix-2 butterflies are merged with the on-the-fly permutation. For the last two layers of radix-2 butterflies, we use `ext` if the root is not a power of x^{16}. Algorithm 15.7 is an example for the radix-2 butterfly with the symbolic root x^2. For the last layer of radix-2 butterflies, we merge the sign-extension and add-sub pairs into the sequence `saddl`, `saddl2`, `ssubl`, `ssubl2`. We then apply one layer of radix-3 butterflies from [DV78a, Equation 8].

- For the input polynomial drawn from $\mathbb{Z}_q[x]/\langle x^{761} - x - 1 \rangle$, we use the 16-bit form and perform one layer of radix-3 butterflies followed by five layers of radix-2 butterflies. This implies only 1536 coefficients are involved in radix-3 butterflies instead of 3072 as for the other input polynomial with coefficients drawn from $\{-1, 0, 1\}$. Concretely, we apply one layer of radix-3 butterflies and two layers of radix-2 butterflies followed by one layer of Barrett reductions while permuting implicitly for Schönhage and Good–Thomas. Then, we perform three layers of radix-2 butterflies and another layer of Barrett reductions. The instructions are similar to Algorithm 15.7, except we store a polynomial with four registers with specifier `.8H`.

Algorithm 15.7 Radix-2 butterfly with symbolic root x^2.

Input: Size-32 8-bit polynomials $a = \texttt{a0} + \texttt{a1}x^{16}, b = \texttt{b0} + \texttt{b1}x^{16}$, where $\texttt{a0}, \texttt{a1}, \texttt{b0}, \texttt{b1}$ are SIMD registers containing:

$$\begin{cases} \texttt{a0} & = a_0 || \cdots || a_7, \\ \texttt{a1} & = a_8 || \cdots || a_{15}, \\ \texttt{b0} & = b_0 || \cdots || b_7, \\ \texttt{b1} & = b_8 || \cdots || b_{15}. \end{cases}$$

Output: $\texttt{a0} + \texttt{a1}x^{16} = (a + bx^2) \bmod (x^{32}+1), \texttt{b0} + \texttt{b1}x^{16} = (a - bx^2) \bmod (x^{32}+1)$

```
ext   v0.16B, b0.16B, b1.16B, #14   ▷ $\texttt{v0} = b_{14} || \cdots || b_{29}$
neg   b1.16B, b1.16B
ext   v1.16B, b1.16B, b0.16B, #14   ▷ $\texttt{v1} = (-b_{30}) || (-b_{31}) || b_0 || \cdots || b_{13}$
sub   b0.16B, a0.16B, v0.16B
sub   b1.16B, a1.16B, v1.16B ▷ $\texttt{b0} + \texttt{b1}x^{16} = (a - x^2 b) \bmod (x^{32}+1)$
add   a0.16B, a0.16B, v0.16B
add   a1.16B, a1.16B, v1.16B ▷ $\texttt{a0} + \texttt{a1}x^{16} = (a + x^2 b) \bmod (x^{32}+1)$
```

Nussbaumer vs Bruun. Next, we review efficient polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^{32}+1 \rangle$. [BBCT22] applied Nussbaumer to $\mathbb{Z}_q[x]/\langle x^{64}+1 \rangle$, and the same approach results in 8 polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^8+1 \rangle$ (cf. Table 5.2). [HLY24] applied Bruun's FFT resulting in multiplications in rings $\mathbb{Z}_q[x]/\langle x^8 + \alpha x^4 + 1 \rangle$ for 4 different α's. Concretely,

$$\begin{aligned} x^{32}+1 &= (x^{16} + 1229x^2 + 1)(x^{16} - 1229x^2 + 1) \\ &= (x^8 + 58x^4 + 1)(x^8 - 58x^4 + 1)(x^8 + 2116x^4 + 1)(x^8 - 2116x^4 + 1) \end{aligned}$$

in $\mathbb{Z}_{4591}[x]$, and we apply $\mathbf{Bruun}_{1229,1}$ followed by $\mathbf{Bruun}_{58,1}$ and $\mathbf{Bruun}_{2116,1}$. See Table 15.6 for comparisons between Nussbaumer and Bruun for polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^{32}+1 \rangle$. Overall, we have 384 size-8 polynomial multiplications of the form $\mathbb{Z}_q[x]/\langle x^8 \pm \alpha x^4 + 1 \rangle$ after applying truncated Schönhage, Good–Thomas, and Nussbaumer.

Table 15.6: Approaches for multiplying polynomials in $\mathbb{Z}_q[x]/\langle x^{64}+1\rangle$ and $\mathbb{Z}_q[x]/\langle x^{32}+1\rangle$.

Approach	Domain	Image	Twiddle factors
Nussbaumer [BBCT22]	$\frac{\mathbb{Z}_q[x]}{\langle x^{64}+1\rangle}$	$\left(\frac{\mathbb{Z}_q[x]}{\langle x^{8}+1\rangle}\right)^{16}$	x^i
Nussbaumer	$\frac{\mathbb{Z}_q[x]}{\langle x^{32}+1\rangle}$	$\left(\frac{\mathbb{Z}_q[x]}{\langle x^{8}+1\rangle}\right)^{8}$	x^{2i}
Bruun	$\frac{\mathbb{Z}_q[x]}{\langle x^{32}+1\rangle}$	$\prod_{i=0,1}\prod\frac{\mathbb{Z}_q[x]}{\langle x^{8}\pm\alpha_i x^4+1\rangle}$	Elements in $\mathbb{Z}_q$.

15.4.1.4 Approach 4: Truncated Rader-17, Good–Thomas, and TMVP

The last approach applies truncated Rader-17, Good–Thomas, and small-dimensional Toeplitz matrix-vector products [Hwa24c], and reduces the number of size-8 polynomial multiplications to 192. There are two main observations as follows.

- As $\mathbb{Z}_{4591}$ contains a principal 17th root of unity, we multiply the polynomials in $\mathbb{Z}_q[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle$.
- In Armv8-A Neon, we can multiply polynomials in $\mathbb{Z}_q[x]\Big/\Big\langle x^{2^k}-\zeta\Big\rangle$ as a Toeplitz matrix-vector product with vector-by-scalar multiplication instructions efficiently (cf. Section 7.3).

Truncated Rader-17. We start with $\mathbb{Z}_q[x]/\langle\Phi_{17}\left(x^{96}\right)\rangle$, and apply truncated Rader-17 as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle\Phi_{17}\left(x^{96}\right)\rangle}\cong\left(\prod\frac{\mathbb{Z}_q[x]}{\langle x^{16}\pm 1\rangle}\right)^{48}.$$

Since the resulting polynomial rings have size 16, the transformation is evidently vectorization-friendly. We detail below the construction and justify its vectorization-friendliness.

Vectorization-friendliness of the truncated Rader's FFT. Let η_0 : $\mathbb{Z}_q^{16}\to\mathbb{Z}_q^{16}$ be the module map implementing the permutation and cyclic convolution parts of the truncated Rader-17. The truncated Rader-17 is implemented as $\texttt{mul}_0\circ\eta_0$ with $\texttt{mul}_0 := (a_i)_{i=0,\ldots,15}\mapsto\left(\omega_{17}^{-(i+1)}a_i\right)_{i=0,\ldots,15}$. We tensor the composition $\mathtt{mul}_0 \circ \eta_0$ by I_{96} and replace the polynomial modulus $\Phi_{17}(x)$ by $\Phi_{17}\left(x^{96}\right)$. This gives us

$$\frac{\mathbb{Z}_q[x]}{\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle} \cong \prod_{i=0}^{15} \frac{\mathbb{Z}_q[x]}{\left\langle x^{96} - \omega_{17}^{i+1}\right\rangle}.$$

We then twist all the rings to the cyclic ones via the product map $\mathtt{twist}_0 := \prod_{i=0}^{15}\left(x \mapsto \omega_{17}^{14(i+1)}x\right)$ where $\omega_{17} = \omega_{17}^{1344} = \left(\omega_{17}^{14}\right)^{96}$. To sum up, we implement

$$\frac{\mathbb{Z}_q[x]}{\left\langle \Phi_{17}\left(x^{96}\right)\right\rangle} \cong \left(\frac{\mathbb{Z}_q[x]}{\left\langle x^{96} - 1\right\rangle}\right)^{16}$$

with

$$\mathtt{twist}_0 \circ \left(\left(\mathtt{mul}_0 \circ \eta_0\right) \otimes I_{96}\right),$$

which is obviously vectorization friendly.

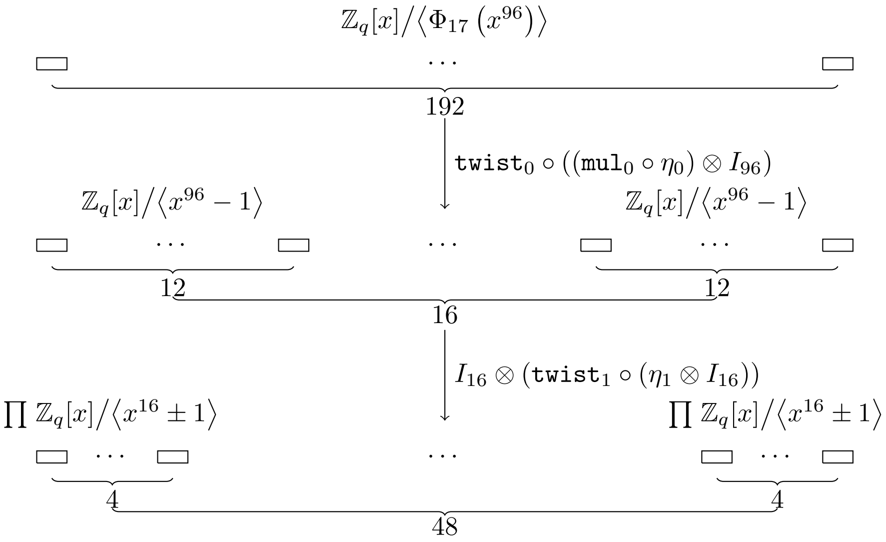


Figure 15.1: Overview of the correspondence between algebraic maps and 128-bit SIMD register view in Armv8-A Neon. Each rectangles holds $\frac{128}{16} = 8$ coefficients and is loaded to a SIMD register. Similar justification of vectorization-friendliness holds in AVX2 with 256-bit SIMD registers.

Vectorization-friendliness of Good–Thomas FFT. Next, we turn the ring $\mathbb{Z}_q[x]/\langle x^{96}-1\rangle$ into $\left(\prod \mathbb{Z}_q[x]/\langle x^{16}\pm 1\rangle\right)^3$ by applying Good–Thomas FFT and twisting. Let η_1 be the map implementing the Good–Thomas FFT of dimension 3×2, and $\texttt{twist}_1$ be the map twisting the product ring into $\left(\prod \mathbb{Z}_q[x]/\langle x^{16}\pm 1\rangle\right)^3$. Then, $\texttt{twist}_1 \circ (\eta_1 \otimes I_{16})$ implements

$$\frac{\mathbb{Z}_q[x]}{\langle x^{96}-1\rangle} \cong \left(\prod \frac{\mathbb{Z}_q[x]}{\langle x^{16}\pm 1\rangle}\right)^3 .$$

Since there are 16 copies of $\mathbb{Z}_q[x]/\langle x^{96}-1\rangle$, we have

$$I_{16} \otimes (\texttt{twist}_1 \circ (\eta_1 \otimes I_{16})) = (I_{16} \otimes \texttt{twist}_1) \circ (I_{16} \otimes \eta_1 \otimes I_{16})$$

as the overall transformation. Obviously, this is vectorization friendly.

For a more illustrative explanation of how polynomials are mapped to 128-bit registers in Armv8-A Neon, see the workflow in Figure 15.1 where each rectangles represents a 128-bit register. Similar justification holds for 256-bit registers since we are right-tensoring by I_{16}.

Toeplitz matrix-vector multiplication for $\mathbb{Z}_q[x]/\langle x^{16}\pm 1\rangle$. For polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^{16}\pm 1\rangle$, since each SIMD registers in Armv8-A Neon holds eight coefficients, we split $\mathbb{Z}_q[x]/\langle x^{16}-1\rangle$ into $\prod \mathbb{Z}_q[x]/\langle x^8\pm 1\rangle$ with radix-2 Cooley–Tukey, and $\mathbb{Z}_q[x]/\langle x^{16}+1\rangle$ into $\left(\mathbb{Z}_q[x]/\langle x^8+1\rangle\right)^3$ with striding Karatsuba. Finally, we compute the products in $\mathbb{Z}_q[x]/\langle x^8\pm 1\rangle$ as Toeplitz matrix-vector products with instructions `ext`, `mul`, `mla`, and `mls`.

15.4.1.5 Performance

Table 15.7 summarizes the performance of the Armv8-A Neon implementations. It should be noted that [HLY24] proposed three approaches: the Good–Thomas with coefficient ring switching (Approach 1, cf. Section 15.4.1.1), the Schönhage, Good–Thomas, and Bruun approach (Approach 3, cf. Section 15.4.1.3), and the Rader-17 approach computing in $\mathbb{Z}_{4591}[x]/\langle x^{1632}-1\rangle$. We skip the description of the Rader-17 approach by [HLY24] since it was further optimized into the truncated Rader-17 approach (Approach 4, cf. Section 15.4.1.4) by [Hwa24c]. See Table 15.7 for a summary of the performance.

Good–Thomas with coefficient ring switching vs Schönhage, Good–Thomas, and Bruun. Comparing the Good–Thomas with coefficient ring switching with the Schönhage, Good–Thomas, and Bruun over $\mathbb{Z}_{4591}$, the overall performance of Good–Thomas with radix-3 and radix-2 butterflies is faster than the Schönhage, Good–Thomas, and Bruun over $\mathbb{Z}_{4591}$ since there are too many small-dimensional polynomial multiplications in the Schönhage, Good–Thomas, and Bruun over $\mathbb{Z}_{4591}$.

Table 15.7: Performance cycles of polynomial multiplications over $\mathbb{Z}_q$ in `sntrup761` with Armv8-A Neon on Cortex-A72 and Firestorm. GT stands for Good–Thomas with coefficient ring switching, Schönhage stands for Schönhage, Good–Thomas, and Bruun (cf. Section 15.4.1.3, Rader-17 stands for Rader-17, Good–Thomas, and Bruun (cf. [HLY24]); and truncated Rader-17 stands for truncated Rader-17, Good–Thomas, and Toeplitz matrix-vector product (cf. Section 15.4.1.4). `mulcore` stands for polynomial multiplication in $\mathbb{Z}_{4591}[x]$ and `polymul` stands for polynomial multiplication in $\mathbb{Z}_{4591}[x]/\langle x^{761} - x - 1\rangle$.

Work	`mulcore`	`polymul`
Cortex-A72		
[HLY24]* (GT)	40 901	46 847
[HLY24]* (Schönhage)	51 613	52 048
[HLY24]* (Rader-17)	37 215	40 950
[Hwa24c]* (truncated Rader-17)	**29 925**	**33 787**
Firestorm		
[HLY24]* (GT)	8 380	9 376
[HLY24]* (Schönhage)	11 706	11 857
[HLY24]* (Rader-17)	8 101	8 912
[Hwa24c]* (truncated Rader-17)	**6 506**	**6 929**

*Benchmark of this thesis.

Good–Thomas with coefficient ring switching vs Rader-17. If we replace the Schönhage with Rader-17 over $\mathbb{Z}_{4591}$, the resulting Rader-17, Good–Thomas, and Bruun approach outperforms the Good–Thomas with coefficient ring switching. The main reason is that there are only half of the small-dimensional polynomial multiplications compared to the Schönhage, Good–Thomas, and Bruun approach.

Rader-17 vs truncated Rader-17. [Hwa24c] further replaced Rader-17 with truncated Rader-17 followed by twisting, removed Bruun, and computed small-dimensional cyclic/negacyclic convolutions with radix-2 Cooley–Tukey and vector-by-scalar multiplication instructions. This is the state-of-the-art implementation. The improvements come from the use of vector-by-scalar multiplication instructions avoiding the interleaving of polynomials, the replacement of $\mathbb{Z}_q[x]/\langle x^{1632}-1\rangle$ by $\mathbb{Z}_q[x]/\langle \Phi_{17}\left(x^{96}\right)\rangle$ removing of the leftover small-dimensional polynomial multiplication, and the Neon-optimized polynomial reduction modulo $x^{761}-x-1$ based on the AVX2 counterpart [BBC+20].

15.4.2 Polynomial Inversion

For polynomial inversion in $\mathbb{Z}_3[x]/\langle x^{761}-x-1\rangle$, we apply the bitsliced arithmetic by [BBC+20] with 128-bit `q` registers in Armv8-A Neon. Table 15.8 summarizes the performance of the old bitsliced approach by [HLY24], the asymptotically faster constant-time GCD by [JWYC24], and the new bitsliced approach based on the circuits by [BBC+20].

Table 15.8: Performance cycles of polynomial inversion in $\mathbb{Z}_3[x]/\langle x^{761}-x-1\rangle$ with Armv8-A Neon on Cortex-A72 and Firestorm.

Work	Cortex-A72	Firestorm
[HLY24]*	587 411	197 222
[JWYC24]	531 825	154 286
This thesis	**374 389**	**128 901**

*Benchmark of this thesis.

15.4.3 Scheme

For the overall performance of `sntrup761` on Cortex-A72 and Firestorm with Armv8-A Neon, this thesis integrates the polynomial multiplications by [HLY24, Hwa24c], the bitsliced polynomial inversion in $\mathbb{Z}_3[x]/\langle x^{761}-x-1\rangle$ by [HLY24], and the new bitsliced polynomial inversion in $\mathbb{Z}_3[x]/\langle x^{761}-x-1\rangle$ based on the circuits by [BBC+20]. Table 15.9 summarizes the performance of the Armv8-A Neon implementations of the parameter set `sntrup761` on Cortex-A72 and Firestorm.

Table 15.9: Performance cycles of `sntrup761` with Armv8-A Neon on Cortex-A72 and Firestorm.

Work	**K**	**E**	**D**
Cortex-A72			
[HLY24]* (GT)	6 658 781	158 989	182 769
[HLY24]* (Schönhage)	6 655 488	162 240	193 993
[HLY24]* (Rader)	6 611 131	150 496	158 711
[Hwa24c]* (truncated Rader)	6 587 485	141 303	135 169
This thesis (truncated Rader)	**6 275 953**	**140 329**	**134 647**
Firestorm			
[HLY24]* (GT)	1 799 422	64 336	45 072
[HLY24]* (Schönhage)	1 805 264	66 523	52 515
[HLY24]* (Rader)	1 798 612	63 848	43 897
[Hwa24c]* (truncated Rader)	1 741 916	61 813	38 120
This thesis (truncated Rader)	**1 677 438**	**61 643**	**37 738**

*Benchmark of this thesis.

15.5 Reviewing and Improving AVX2 Implementations

This section goes through the AVX2 implementations of `sntrup761`. As the AVX2-optimized implementation is similar to the Armv8-A Neon, we only highlight the differences.

15.5.1 Polynomial Multiplication

For the AVX2-optimized polynomial multiplication in `sntrup761`, the fastest approach is also based on truncated Rader-17 and Good–Thomas [Hwa24c]. The only differences to the Armv8-A Neon implementation are the polynomial multiplications in $\left(\mathbb{Z}_q[x]/\langle x^{16} \pm 1\rangle\right)^{48}$. In AVX2, we do not have the convenient `ext` instruction extracting consecutive bytes as in Neon, and a permutation-friendly transformation is required for efficient vectorization.

Permutation-friendly transformation. Since the goal is to interleave 16 polynomials drawn from polynomial rings with the same shape of computation, we show how to map the polynomial multiplications in the product ring $\left(\prod \mathbb{Z}_q[x]/\langle x^{16} \pm 1\rangle\right)^{16}$ to vector arithmetic. We perform an even-odd permutation over 16-tuples resulting $\left(\mathbb{Z}_q[x]/\langle x^{16} - 1\rangle\right)^{16} \times \left(\mathbb{Z}_q[x]/\langle x^{16} + 1\rangle\right)^{16}$ followed by two copies of T_{256}. This gives us the map

$$(I_2 \otimes T_{256})\,(\texttt{EvenOdd}_{32} \otimes I_{16})$$

where $\texttt{EvenOdd}_{32}$ moves the even indices to the first half and the odd indices to the second half. See Figure 15.2 for an illustration. The overall interleaving matrix for $\left(\prod \mathbb{Z}_q[x]/\langle x^{16} \pm 1\rangle\right)^{48}$ can be written as:

$$(I_6 \otimes T_{256})\,(I_3 \otimes \texttt{EvenOdd}_{32} \otimes I_{16})$$

which is permutation-friendly. For the follow-up polynomial multiplications, we apply Cooley–Tukey to $\mathbb{Z}_q[x]/\langle x^{16} - 1\rangle \cong \prod \mathbb{Z}_q[x]/\langle x^8 \pm 1\rangle$ and Bruun to $\mathbb{Z}_q[x]/\langle x^{16} + 1\rangle \cong \mathbb{Z}_q[x]/\langle x^8 \pm \sqrt{2}x^4 + 1\rangle$ followed by Karatsuba defined over SIMD registers.

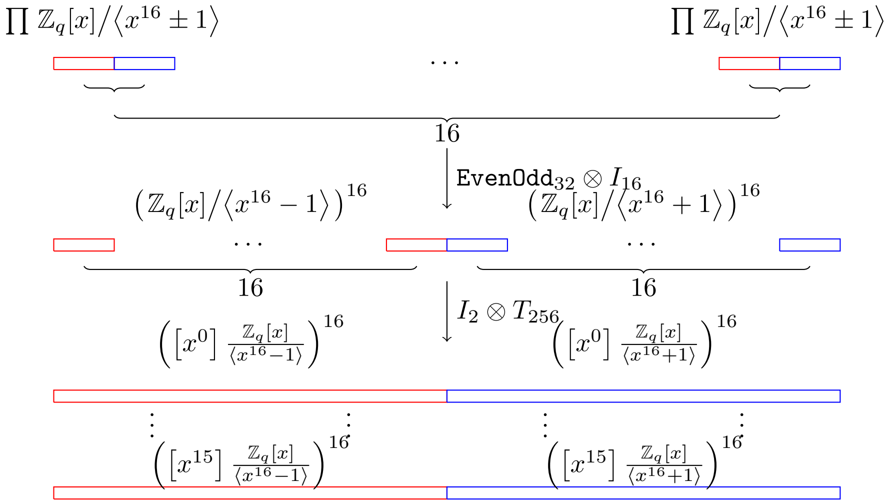


Figure 15.2: Overview of AVX2 permutation for $\left(\mathbb{Z}_q[x]/\langle x^{16} \pm 1\rangle\right)^{16}$. Same idea applies to $\left(\mathbb{Z}_q[x]/\langle x^{16} \pm 1\rangle\right)^{48}$ since $48 = 3 \cdot 16$. Each rectangles represents a 16-tuple stored in a 256-bit SIMD register in AVX2.

As illustrated in Table 15.10, the resulting polynomial multiplication is significantly faster than the well-optimized polynomial multiplication by [BBCT22] on Haswell and Skylake.

Table 15.10: Performance cycles of polynomial multiplications over $\mathbb{Z}_q$ in `sntrup761` with AVX2 on Haswell and Skylake.

Operation	Work	Haswell	Skylake
`mulcore` $(\mathbb{Z}_{4591}[x])$	[BBCT22]*	23 460	20 070
	[Hwa24c]	**12 336**	**9 778**
`polymul` $\left(\frac{\mathbb{Z}_{4591}[x]}{\langle x^{761}-x-1\rangle}\right)$	[BBCT22]*	25 356	21 364
	[Hwa24c]	**12 760**	**9 876**

*Reported by [Hwa24c].

15.5.2 Scheme

For the performance of the overall scheme, this thesis reports the benchmarks by [Hwa24c]. They integrated their implementation into the package `libsntrup761` with version `20210608` provided by [BBCT22], and reported the the amortized cost of batch key generation with batch size 32. Additionally, they also integrated their implementation into the package `supercop` with version `20230530`, and reported the performance of encapsulation and decapsulation. See Table 15.11 for the overall performance.

Table 15.11: Performance cycles of `sntrup761` with AVX2 on Haswell and Skylake.

Operation	Work	Haswell	Skylake
Batch key generation	[BBCT22]*	154 552	129 159
	[Hwa24c]	**136 003**	**118 939**
Encapsulation	SUPERCOP*	47 464	40 653
	[Hwa24c]	**44 108**	**36 486**
Decapsulation	SUPERCOP*	56 064	47 387
	[Hwa24c]	**50 080**	**41 070**

*Reported by [Hwa24c].

Chapter 16

Saber

16.1 Specification

Saber is a lattice-based KEM [DKRV20] in the 3rd round of the NIST PQC Standardization, and is based on the hardness of M-LWR. For simplicity, we go through the underlying PKE scheme and refer to [DKRV20] for the full specification of the KEM. Let $q = 8192, n = 256, \ell, \mu, p$, and T be integers where ℓ, p, μ, and T vary between parameter sets. Parameters q and n determine the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, and parameter ℓ determines the module $R_q^{\ell\times\ell}$ over R_q. Parameter μ determines the centered binomial distribution for the secret values, and parameters p and T determine the rounding of the ciphertexts. See Table 16.1 for a summary of the parameter sets.

Table 16.1: Saber parameter sets.

Parameter set	q	n	ℓ	μ	$p = 2^{\epsilon_p}$	$T = 2^{\epsilon_T}$
`lightsaber`	8192	256	2	10	1024	8
`saber`	8192	256	3	8	1024	16
`firesaber`	8192	256	4	6	1024	64

Key generation. In the key generation, we generate a matrix $\mathbf{A} \in R_q^{\ell\times\ell}$ from the seed $\mathrm{seed}_{\mathbf{A}}$, sample a vector $\mathbf{s} \in R_q^{\ell}$ from the centered binomial distribution with μ as the parameter, compute $\mathbf{A}^{\top}\mathbf{s}$, and round it to the vector $\mathbf{b} \in R_q^{\ell}$. The public key is defined as $(\mathrm{seed}_{\mathbf{A}}, \mathbf{b})$, and the secret key is defined as $\mathbf{s}$.

Algorithm 16.1 Saber PKE key generation.

```
Output:
    $\begin{cases} \text{Public key} & pk & = \texttt{pkEncode}\left(\text{seed}_{\mathbf{A}}, \mathbf{b}\right). \\ \text{Secret key} & sk & = \texttt{skEncode}\left(\mathbf{s}\right). \end{cases}$
$\text{seed}_{\mathbf{A}} \leftarrow \{0,1\}^{256}$
$r \leftarrow \{0,1\}^{256}$
$\mathbf{A} \in R_q^{\ell \times \ell} \leftarrow \texttt{ExpandA}\left(\text{seed}_{\mathbf{A}}\right)$
$\mathbf{s} \in R_q^{\ell} \leftarrow \texttt{SampleS}_{\mu}\left(r\right)$
$\mathbf{b} \leftarrow \texttt{Round}_{p,q}\left(\mathbf{A}^{\top}\mathbf{s}\right)$
$pk = \texttt{pkEncode}\left(\text{seed}_{\mathbf{A}}, \mathbf{b}\right)$
$sk = \texttt{skEncode}\left(\mathbf{s}\right)$
```

Encryption. For the encryption, we sample a vector $\mathbf{s} \in R_q^{\ell}$ from the centered binomial distribution with μ as the parameter. We then decode the message m to a polynomial and compute $c_m = \texttt{Round}_{T,p}\left(\mathbf{b}^{\top}\mathbf{s} - 2^{e_p - 1}\texttt{ByteDecode}_1(m)\right)$ and $\mathbf{u} = \texttt{Round}_{p,q}\left(\mathbf{A}\mathbf{s}\right)$. The ciphertext ct is defined as the encoding of $\mathbf{u}$ and c_m. See Algorithm 16.2 for an illustration.

Algorithm 16.2 Saber PKE encryption.

```
Input:
    $\begin{cases} \text{Public key} & pk & = \texttt{pkEncode}\left(\text{seed}_{\mathbf{A}}, \mathbf{b}\right). \\ \text{Message} & m & \in \{0,1\}^{256}. \\ \text{Randomness} & r & \in \{0,1\}^{256}. \end{cases}$
Output: Ciphertext $ct = (c_1, c_2)$.
$(\text{seed}_{\mathbf{A}}, \mathbf{b}) \leftarrow \texttt{pkDecode}(pk)$
$\mathbf{A} \in R_q^{\ell \times \ell} \leftarrow \texttt{ExpandA}\left(\text{seed}_{\mathbf{A}}\right)$
$\mathbf{s} \in R_q^{\ell} \leftarrow \texttt{SampleS}_{\mu}\left(r\right)$
$\mathbf{u} \leftarrow \texttt{Round}_{p,q}\left(\mathbf{A}\mathbf{s}\right)$
$v \leftarrow \mathbf{b}^{\top}\mathbf{s}$
$c_m \leftarrow \texttt{Round}_{T,p}\left(v - 2^{e_p - 1}\texttt{ByteDecode}_1(m)\right)$
$c_1 \leftarrow \texttt{ByteEncode}_{e_p}\left(\mathbf{u}\right)$
$c_2 \leftarrow \texttt{ByteEncode}_{e_T}\left(c_m\right)$
$ct = (c_1, c_2)$
```

Decryption. For decryption, we decode the ciphertext ct into a vector $\mathbf{u} \in R_q^{\ell}$ and $c_m \in R_q$, compute $v' = \mathbf{u}^{\top}\mathbf{s}$ and $\texttt{Round}_{T,p}\left(v' - 2^{e_p - e_T}c_m\right)$. The message m is retrieved by encoding the result. See Algorithm 16.3 for an illustration.

Algorithm 16.3 Saber PKE decryption.

```
Input:
    $\begin{cases} \text{Ciphertext} & ct & = (c_1, c_2). \\ \text{Secret key} & sk & = \texttt{skEncode}(\mathbf{s}). \end{cases}$
Output: Message $m \in \{0,1\}^{256}$.
1: $\mathbf{s} \leftarrow \texttt{skDecode}(sk)$
2: $\mathbf{u} \leftarrow \texttt{ByteDecode}_{e_p}(c_1)$
3: $c_m \leftarrow \texttt{ByteDecode}_{e_T}(c_2)$
4: $v' \leftarrow \mathbf{u}^\top \mathbf{s}$
5: $m \leftarrow \texttt{ByteEncode}_1(\texttt{Round}_{T,p}(v' - 2^{e_p - e_T} c_m))$
```

16.2 Optimization Guide for Polynomial Arithmetic

16.2.1 Polynomial Multiplication

For multiplying two polynomials in $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$, since q is a power of two in Saber and we do not have native radix-2 Cooley–Tukey FFT over $\mathbb{Z}_q$, we have to injectively map the polynomial ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ to a larger polynomial ring with efficient polynomial multiplication.

Toom–Cook. Let $k > 1$ be an integer, $\mathcal{I} = \{0, \dots, k-1\}$, and $\{s_i | i \in \mathcal{I}\} \subset \mathbb{Q} \cup \{\infty\}$. Toom-$k$ works as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^n + 1\rangle} \to \frac{\mathbb{Z}_q[x]}{\langle \prod_{i \in \mathcal{I}} (x^{n/k} - s_i)\rangle} \to \prod_{i \in \mathcal{I}} \frac{2^{-\mathbb{Z}_{\geq 0}} \mathbb{Z}_q[x]}{\langle x^{n/k} - s_i \rangle}.$$

As for $2^{-\mathbb{Z}_{\geq 0}} \mathbb{Z}_q[x] \Big/ \Big\langle x^{n/k} - s_i \Big\rangle$, we recursively apply Toom–Cook. Since $n = 256$ is a power of two in Saber, a common way is to apply a series of Toom-4's and Karatsuba's (Toom-2). While applying Toom-4, one has to keep track of the precision as elements in $2^{-\mathbb{Z}_{\geq 0}} \mathbb{Z}_q$ are scaled to integers (cf. Section 5.1). Recall that $q = 2^{13}$ in Saber. If we work with 16-bit arithmetic entirely, then we can only adjoin inverses $2^{-1}, 2^{-2}, 2^{-3}$. This restricts how many Toom-4's one can apply. In the literature [KRS19, MKV20], the state-of-the-art Toom–Cook approach consists of one layer of Toom-4 followed by two layers of Karatsuba.

Striding Toom–Cook. One can exploit the structure of the polynomial modulus with striding Toom–Cook as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^n+1\rangle} \to \frac{\mathbb{Z}_q[y]\Big/\Big\langle y^{n/k}+1\Big\rangle\,[x]}{\langle x^k-y\rangle} \to \frac{\mathbb{Z}_q[y]\Big/\Big\langle y^{n/k}+1\Big\rangle\,[x]}{\left\langle\prod_{i\in\mathcal{I}}(x-s_i)\right\rangle}.$$

We similarly map $\mathbb{Z}_q$ to $2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}_q$, and recursively apply striding Toom–Cook while tracking the precision. This was implemented in [BMK+21] on the M-profile Vector Extension (MVE) targeting the Armv8-M architecture.

Toeplitz-TC. We recall that for a polynomial $\boldsymbol{a} \in \mathbb{Z}_q[x]/\langle x^n+1\rangle$, the associated multiplication map $\boldsymbol{a} \mapsto \boldsymbol{ab} : \mathbb{Z}_q[x]/\langle x^n+1\rangle \to \mathbb{Z}_q[x]/\langle x^n+1\rangle$ can be phrased as an application of a Toeplitz matrix constructed from the coefficients of $\boldsymbol{a}$. Suppose we have a composition of Toom–Cook's multiplying two size-n polynomials, then this composition can be turned into a series of Toeplitz matrix-vector multiplications implementing a Toeplitz matrix-vector product with matrix dimension $n \times n$ (cf. Section 6.5). [IKPC20] was the first applying the idea to Saber.

Schönhage/Nussbaumer. We can also apply radix-2 Schönhage and Nussbaumer over $\mathbb{Z}_q$. We explicit the Nussbaumer for the case $n = 256$ in Saber as follows:

$$\frac{\mathbb{Z}_q[x]}{\langle x^{256}+1\rangle} \to \frac{\mathbb{Z}_q[y]/\left\langle y^{16}+1\right\rangle\,[x]}{\langle x^{16}-y\rangle} \to \frac{\mathbb{Z}_q[y]/\left\langle y^{16}+1\right\rangle\,[x]}{\langle x^{32}-1\rangle}.$$

After replacing $\mathbb{Z}_q$ by $2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}_q$, we can choose between Schönhage/Nussbaumer, Toeplitz-TC, striding Toom–Cook for $2^{-\mathbb{Z}_{\geq 0}}\mathbb{Z}_q[y]/\left\langle y^{16}+1\right\rangle$.

Coefficient ring switching and radix-2 Cooley–Tukey FFT. The last approach is to switch to a coefficient ring defining a radix-2 Cooley–Tukey FFT. Since one of the polynomial operands has coefficients with absolute values bounded by η, the coefficients of the polynomial product over $\mathbb{Z}$ have absolute values bounded by $\frac{q\eta}{4}$. This implies we can choose a new modulus $q' > \frac{q\eta}{2}$ containing a principal 2^{m+1}th root of unity and apply radix-2 size-2^m Cooley–Tukey FFT to $\mathbb{Z}_{q'}[x]/\langle x^n+1\rangle$.

16.2.2 Matrix-Vector Multiplication

Cost analysis with algebra monomorphisms. In Saber, we have to apply the public matrix $\mathbf{A}$ to the secret vector $\mathbf{s}$ over $\mathbb{Z}_q[x]/\langle x^n+1\rangle$. Suppose we find an algebra homomorphism f implementing $\boldsymbol{ab} = f^{-1}(f(\boldsymbol{a})f(\boldsymbol{b}))$ for polynomials $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. We can naturally extend f to matrices over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$: for a matrix $\mathbf{A}$, $f(\mathbf{A})$ is defined as the matrix of the same dimension where each entry is replaced by its image under f. We compute $\mathbf{As}$ as

$$f^{-1}(f(\mathbf{A})f(\mathbf{s})).$$

We recall the cost analysis by [Hwa22] as follows. Define $\mathcal{C}$ as the function mapping a homomorphism to its computational cost. For an $\ell \times \ell$ matrix $\mathbf{A}$ and an $\ell \times 1$ vector $\mathbf{s}$ over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$, the cost of $f^{-1}(f(\mathbf{A})f(\mathbf{s}))$ is

$$\left(\ell^2 + \ell\right)\mathcal{C}(f) + \ell^2\mathcal{C}\left(\mathtt{mul}_f\right) + \ell\mathcal{C}\left(f^{-1}\right)$$

where $\mathtt{mul}_f$ is the ring multiplication with optional accumulation in the image of f.

Cost analysis with bilinear maps. We further the cost analysis with bilinear maps. Suppose now we have module homomorphisms $f_{\mathtt{I}}, f_{\mathtt{L}}, f_{\mathtt{R}}$, and a bilinear map g implementing

$$\boldsymbol{ab} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}(\boldsymbol{a}), f_{\mathtt{R}}(\boldsymbol{b})\right)\right)$$

for polynomials $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. The maps naturally generalize to the matrix-vector multiplication

$$\mathbf{As} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}\left(\mathbf{A}\right)\right), f_{\mathtt{R}}(\mathbf{s})\right)$$

with the computational cost

$$\ell^2\mathcal{C}\left(f_{\mathtt{L}}\right) + \ell\mathcal{C}\left(f_{\mathtt{R}}\right) + \ell^2\mathcal{C}\left(g\right) + \ell\mathcal{C}\left(f_{\mathtt{I}}\right)$$

Obviously, this generalizes the cost analysis with algebra monomorphisms.

Optimizing matrix-vector multiplication with homomorphisms. To optimize the the matrix-vector multiplication, we first find an algebra monomorphism f computing $\mathbf{As} = f^{-1}(f(\mathbf{A})f(\mathbf{s}))$. If $\mathcal{C}\left(f^{-1}\right) > \mathcal{C}(f)$, we dualize the maps and convert the polynomial multiplications into Toeplitz matrix-vector multiplications. As the coefficient ring is a commutative ring, we have two ways rewriting a product $\boldsymbol{ab}$ of polynomials as a Toeplitz matrix-vector product: either as $\boldsymbol{ab} = f_{\mathtt{I}}\left(g_{\mathtt{L}}\left(f_{\mathtt{L}}(\boldsymbol{a}), f_{\mathtt{R}}(\boldsymbol{b})\right)\right)$ or as $\boldsymbol{ab} = f_{\mathtt{I}}\left(g_{\mathtt{R}}\left(f_{\mathtt{R}}(\boldsymbol{a}), f_{\mathtt{L}}(\boldsymbol{b})\right)\right)$ for $f_{\mathtt{I}} = f_{\mathtt{L}} = f^*$, $f_{\mathtt{R}} = f^{-1*}$, and $g_{\mathtt{L}}, g_{\mathtt{R}}$ bilinear maps composed of small-dimensional Toeplitz matrix-vector multiplications. In the context of polynomial multiplication, there are no differences in computational cost. But, when generalizing to matrix-vector multiplications, they amount to different computational cost. The former generalizes to

$$\mathbf{A}\mathbf{s} = f_{\mathtt{I}}\left(g_{\mathtt{L}}\left(f_{\mathtt{L}}\left(\mathbf{A}\right), f_{\mathtt{R}}\left(\mathbf{s}\right)\right)\right)$$

with the computational cost

$$\ell^2 \mathcal{C}\left(f_{\mathtt{L}}\right) + \ell \mathcal{C}\left(g_{\mathtt{R}}\right) + \ell^2 \mathcal{C}\left(g\right) + \ell \mathcal{C}\left(f_{\mathtt{I}}\right)$$

and the latter generalizes to

$$\mathbf{A}\mathbf{s} = f_{\mathtt{I}}\left(g_{\mathtt{R}}\left(f_{\mathtt{L}}\left(\mathbf{s}\right), f_{\mathtt{R}}\left(\mathbf{A}\right)\right)\right)$$

with the computational cost

$$\ell^2 \mathcal{C}\left(f_{\mathtt{R}}\right) + \ell \mathcal{C}\left(g_{\mathtt{L}}\right) + \ell^2 \mathcal{C}\left(g\right) + \ell \mathcal{C}\left(f_{\mathtt{I}}\right).$$

Typically, $\mathcal{C}\left(f^*\right) = \mathcal{C}(f)$, $\mathcal{C}\left(f^{-1*}\right) = \mathcal{C}\left(f^{-1}\right)$, $\mathcal{C}\left(g_{\mathtt{L}}\right) = \mathcal{C}\left(g_{\mathtt{R}}\right)$, and we have $\mathcal{C}\left(f_{\mathtt{R}}\right) = \mathcal{C}\left(f^{-1*}\right) > \mathcal{C}\left(f^*\right) = \mathcal{C}\left(f_{\mathtt{L}}\right)$. Therefore, computing with $\mathbf{A}\mathbf{s} = f_{\mathtt{I}}\left(g_{\mathtt{L}}\left(f_{\mathtt{L}}\left(\mathbf{A}\right), f_{\mathtt{R}}\left(\mathbf{s}\right)\right)\right)$ is more preferable if $\mathcal{C}\left(f^{-1}\right) > \mathcal{C}\left(f\right)$ [HKS24, Section 2.5]. [IKPC20] also mentioned one can cache the images with Toeplitz-TC, but they didn't implement the idea and it was unclear which images they would like to cache.

16.2.3 Inner Product

Assume we already find homomorphisms $f_{\mathtt{I}}, f_{\mathtt{L}}, f_{\mathtt{R}}$, and a bilinear map g implementing the matrix-vector multiplication

$$\mathbf{A}\mathbf{s} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}\left(\mathbf{A}\right), f_{\mathtt{R}}\left(\mathbf{s}\right)\right)\right).$$

We continue with the inner products in encryption (Algorithm 16.2) and decryption (Algorithm 16.3).

Cost analysis of the inner product in decryption. During the decryption, the only polynomial multiplications are the ones in the inner product $\mathbf{u}^T\mathbf{s}$ of vectors $\mathbf{u}, \mathbf{s}$ over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$. We compute as follows

$$\mathbf{u}^T\mathbf{s} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}\left(\mathbf{u}\right)^T, f_{\mathtt{R}}(\mathbf{s})\right)\right)$$

with the computational cost

$$\ell \mathcal{C}\left(f_{\mathtt{L}}\right) + \ell \mathcal{C}\left(f_{\mathtt{R}}\right) + \ell \mathcal{C}\left(g\right) + \mathcal{C}\left(f_{\mathtt{I}}\right).$$

Cost analysis of the inner product in encryption. During the encryption, we have to compute the matrix-vector product $\mathbf{As}$ and the inner product $\mathbf{b}^T\mathbf{s}$ over $\mathbb{Z}_q[x]/\langle x^n+1\rangle$. Since the vector $\mathbf{s}$ is shared among the matrix-vector multiplication and the inner product, we compute $f_{\mathtt{R}}(\mathbf{s})$ during $\mathbf{As}$, store $f_{\mathtt{R}}(\mathbf{s})$ in memory, and load $f_{\mathtt{R}}(\mathbf{s})$ from memory while computing $\mathbf{b}^T\mathbf{s}$ [CHK+21]. Therefore, the cost of $\mathbf{b}^T\mathbf{s} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}\left(\mathbf{b}\right)^T, f_{\mathtt{R}}(\mathbf{s})\right)\right)$ is reduced to

$$\ell\mathcal{C}\left(f_{\mathtt{L}}\right) + \ell\mathcal{C}\left(g\right) + \mathcal{C}\left(f_{\mathtt{I}}\right).$$

16.3 Reviewing and Improving Cortex-M3 Implementations

For the Cortex-M3 implementations of Saber, this thesis compares and benchmarks the implementations of [ACC+21, HKS24].

16.3.1 Polynomial Multiplication

For the polynomial multiplication on Cortex-M3, there are several options for multiplying two polynomials in $\mathbb{Z}_q[x]/\langle x^n+1\rangle$. Since $q = 2^{13}$ is a power of two, we cannot define multiplication-based radix-2 Cooley–Tukey FFT over $\mathbb{Z}_q$. A straightforward approach is Toom–Cook over $\mathbb{Z}_{2^k}$. An alternative is to choose a new NTT-friendly modulus q' with $\frac{q'}{2}$ upper bounding the absolute values of the products over $\mathbb{Z}$. Since one of the operands in the polynomial multiplication has coefficients with absolute values bounded by η, we can choose one or more NTT-friendly moduli whose product upper bounds the results over $\mathbb{Z}$ and compute with NTTs accordingly (cf. Section 16.2). [HKS24] proposed Nussbaumer crafting the principal roots of unity defining a radix-2 Cooley–Tukey FFT and computed the small-dimensional polynomial multiplication with Toeplitz-TC. See Table 16.2 for a summary of the performance.

Table 16.2: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n+1\rangle$ of Saber on Cortex-M3.

Operation	TC	32-bit NTT	16-bit NTT	Nussbaumer
	pqm3**	[ACC+21]*	[ACC+21]*	[HKS24]*
NTT/Hom-M	-	31 707	16 779	15 803
NTT/Hom-V	-	-	-	7 856
Variable-time NTT	-	19 949	-	-
Mul./BiHom	-	8 530	11 933	11 257
NTT^{-1}	-	38 027	19 058	10 881
CRT	-	-	4 638	-
Polymul.	89 590	98 213	69 187	**45 797**

*Benchmark of this thesis.
**Benchmark of this thesis. Based on `https://github.com/mupq/pqm3/commit/a2bea8b1740f6412218e09d17f871791d24d633f`.

32-bit NTT vs 16-bit NTT. For the NTT approach, a 32-bit modulus is sufficient for computing the results over $\mathbb{Z}$. Since 32-bit modular multiplications are slow on Cortex-M3 (cf. Table 9.8), computing 16-bit NTTs over two 16-bit NTT-friendly moduli is more preferable.

16-bit NTT vs Nussbaumer. As for the comparison between the 16-bit NTT approach and the Nussbaumer approach, Nussbaumer is faster as we only need to apply the transformation once and two 16-bit NTTs are required for the 16-bit NTT approach.

Toom–Cook vs Nussbaumer. As for the Toom–Cook approach over $\mathbb{Z}_{2^k}$, Nussbaumer is also faster since Nussbaumer results in a smaller number of small-dimensional polynomial multiplications.

16.3.2 Matrix-Vector Multiplication and Inner Product

For the matrix-vector multiplications over $\mathbb{Z}_q[x]/\langle x^n+1\rangle$, we recall below the cost analysis of the computation. Let $f_{\mathtt{L}}, f_{\mathtt{R}}, f_{\mathtt{I}}$ be module homomorphisms and g be a bilinear map implementing $\boldsymbol{ab} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}(\boldsymbol{a}), f_{\mathtt{R}}(\boldsymbol{b})\right)\right)$. For the matrix-vector multiplication $\mathbf{As}$ with matrix dimension $\ell\times\ell$, we have $\mathbf{As} = f_{\mathtt{I}}\left(g\left(f_{\mathtt{L}}(\mathbf{A}), f_{\mathtt{R}}(\mathbf{s})\right)\right)$ with the computational cost $\ell^2\mathcal{C}\left(f_{\mathtt{L}}\right)+\ell\mathcal{C}\left(f_{\mathtt{R}}\right)+\ell^2\mathcal{C}(g)+\ell\mathcal{C}\left(f_{\mathtt{I}}\right)$ (cf. Section 16.2).

We compare four approaches: (i) Toom–Cook in pqm3, (ii) 32-bit NTT by [ACC+21], (iii) 16-bit NTT by [ACC+21], and (iv) Nussbaumer+Toeplitz-TC by [HKS24]. See Table 16.3 for a summary of the performance.

Table 16.3: Performance cycles of matrix-vector multiplications, inner products in encryptions, and inner products in decryptions of Saber on Cortex-M3.

Operation	Work	`lightsaber`	`saber`	`firesaber`
MV	pqm3**	361 260	815 146	1 451 149
	[ACC+21]* (32-bit)	300 876	573 300	927 003
	[ACC+21]* (16-bit)	199 202	390 762	643 587
	[HKS24]*	**136 462**	**271 605**	**455 527**
IP(Enc)	pqm3	-	-	-
	[ACC+21]* (32-bit)	118 805	159 472	200 126
	[ACC+21]* (16-bit)	82 906	113 562	144 200
	[HKS24]*	**52 479**	**74 885**	**98 328**
IP(Dec)	pqm3**	180 655	271 744	362 819
	[ACC+21]* (32-bit)	182 120	254 430	326 737
	[ACC+21]* (16-bit)	116 350	163 716	211 088
	[HKS24]*	**83 985**	**121 932**	**160 703**

*Benchmark of this thesis.
**Benchmark of this thesis. Based on `https://github.com/mupq/pqm3/commit/a2bea8b1740f6412218e09d17f871791d24d633f`.

Toom–Cook vs others. Since the dominating term of the computational cost of the matrix-vector multiplication is $\mathcal{C}(f_{\mathrm{L}}) + \mathcal{C}(g)$ and 32-bit NTT, 16-bit NTT, and Nussbaumer+Toeplitz-TC approaches amount to smaller numbers of small-dimensional polynomial multiplications compared to Toom–Cook at the cost of somewhat expensive but reasonable transformations, they outperform the Toom–Cook approach.

32-bit NTT vs 16-bit NTT. For the NTT approaches, the primary factor is the cost of modular multiplications. Since 16-bit modular multiplication is at least 2× faster than the constant-time 32-bit modular multiplication and slightly faster than the variable-time 32-bit modular multiplication on Cortex-M3 (cf. Tables 9.8 and 9.9), the 16-bit NTT approach is faster than the 32-bit NTT approach.

16-bit NTT vs Nussbaumer+Toeplitz-TC. As for the comparison between 16-bit NTT and Nussbaumer+Toeplitz-TC, since Nussbaumer+Toeplitz-TC operates entirely over 32-bit registers and amount to small-dimensional polynomial multiplications over $\mathbb{Z}_{2^{32}}$ whose ring multiplication is much faster than the modular multiplications in 16-bit NTTs, Nussbaumer+Toeplitz-TC outperforms the 16-bit NTT approach.

16.3.3 Scheme

For the overall performance of Saber on Cortex-M3, this thesis benchmarks the Toom–Cook in pqm3, the 32-bit NTT and 16-bit NTT approaches by [ACC$^+$21], and the Nussbaumer+Toeplitz approach by [HKS24]. See Table 16.4 for a summary of the performance.

Table 16.4: Performance cycles of Saber on Cortex-M3.

Parameter set	Work	**K**	**E**	**D**
`lightsaber`	pqm3**	625k	894k	1 021k
	[ACC$^+$21]* (32-bit)	512k	696k	795k
	[ACC$^+$21]* (16-bit)	458k	629k	686k
	[HKS24]*	**391k**	**535k**	**567k**
`saber`	pqm3**	1 284k	1 671k	1 861k
	[ACC$^+$21]* (32-bit)	924k	1 167k	1 296k
	[ACC$^+$21]* (16-bit)	847k	1 080k	1 153k
	[HKS24]*	**728k**	**919k**	**962k**
`firesaber`	pqm3**	2 156k	2 650k	2 912k
	[ACC$^+$21]* (32-bit)	1 421k	1 713k	1 881k
	[ACC$^+$21]* (16-bit)	1 327k	1 609k	1 709k
	[HKS24]*	**1 125k**	**1 358k**	**1 424k**

*Benchmark of this thesis.
**Benchmark of this thesis. Based on `https://github.com/mupq/pqm3/commit/a2bea8b1740f6412218e09d17f871791d24d633f`.

16.4 Reviewing Cortex-M4 Implementations

For Cortex-M4 implementations of Saber, this thesis reviews the works [KRS19, IKPC20, CHK$^+$21, ACC$^+$21, BMK$^+$21], and benchmarks the implementations by [IKPC20, ACC$^+$21].

16.4.1 Polynomial Multiplication

For the polynomial multiplications over $\mathbb{Z}_{2^k}$ for Saber on Cortex-M4, there are several works: the Toom–Cook over $\mathbb{Z}_{2^k}$ by [KRS19, IKPC20], the striding Toom–Cook over $\mathbb{Z}_{2^k}$ by [BMK$^+$21], the Toeplitz-TC over $\mathbb{Z}_{2^k}$ by [IKPC20], and the 32-bit NTTs by [CHK$^+$21, ACC$^+$21]. See Table 16.5 for a summary.

Table 16.5: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ of Saber on Cortex-M4.

Approach	Work	Cycle
TC	[KRS19]*	39 124
Toeplitz-TC	[IKPC20]**	28 524
Striding TC	[BMK$^+$21]	34 884
32-bit NTT	[ACC$^+$21]	**23 107**

* Polynomial reduction excluded.
** Benchmark of this thesis.

Toom–Cook vs striding Toom–Cook. For the Toom–Cook by [KRS19, MKV20] and the striding Toom–Cook by [BMK$^+$21], striding Toom–Cook performs faster since it saves some memory operations during interpolations.

Striding Toom–Cook vs Toeplitz-TC. The Toeplitz-TC by [IKPC20] fully exploits the structure of the polynomial ring, and is faster than the striding Toom–Cook by [BMK$^+$21] even though [IKPC20] was already public two years prior to [BMK$^+$21].

Toeplitz-TC vs 32-bit NTT. For the comparisons of Toeplitz-TC and 32-bit NTT, since 32-bit Montgomery multiplication takes only three cycles (cf. Table 9.10), the 32-bit `NTT` and $\texttt{NTT}^{-1}$ are very fast and the resulting polynomial multiplication outperformed the Toeplitz-TC approach. See Table 16.6 for the performance cycles of the operations in the 32-bit NTT approach.

Nussbaumer+Toeplitz-TC vs 32-bit NTT. For Nussbaumer+Toeplitz-TC, since the coefficient ring is replaced by $\mathbb{Z}_{2^{21}}$, we cannot use DSP instructions in Armv7E-M and its performance is mostly the same as Cortex-M3. Therefore, Nussbaumer+Toeplitz-TC approach is not worth trying on Cortex-M4. On the other hand, 32-bit NTTs are very fast due to the powerful

1-cycle long multiplication instructions. The fastest approach is the 32-bit NTT [ACC+21].

Table 16.6: Performance cycles of polynomial multiplications in $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ with 32-bit NTT in Saber on Cortex-M4.

Operation	`NTT`	Mul.	`NTT`$^{-1}$	Polymul.
32-bit NTT, [ACC+21]*	5 861	4 189	7 196	23 107

* Benchmark of this thesis.

16.4.2 Matrix-Vector Multiplication and Inner Product

We compare the following approaches in the literature: Toom–Cook by [KRS19], Toom–Cook with cached evaluation and lazy interpolation by [MKV20], striding Toom–Cook by [BMK+21], Toeplitz-TC by [IKPC20], and 32-bit NTTs by [CHK+21, ACC+21].

Table 16.7: Performance cycles of matrix-vector multiplications, inner products in encryptions, and inner products in decryptions of Saber on Cortex-M4.

Operation	Work	`lightsaber`	`saber`	`firesaber`
MV	[IKPC20]*	121 801	273 412	484 899
	[ACC+21]*	**68 884**	**137 764**	**229 630**
IP(Enc)	[IKPC20]*	-	-	-
	[ACC+21]*	**28 649**	**40 145**	**51 642**
IP(Dec)	[IKPC20]*	60 792	91 141	121 489
	[ACC+21]*	**40 265**	**57 563**	**74 866**

* Benchmark of this thesis.

Toom–Cook, striding Toom–Cook, and Toeplitz-TC. [KRS19] implemented the Toom–Cook approach but they didn't exploit the homomorphic property of Toom–Cook for matrix-vector multiplications and inner products. [MKV20] exploited the homomorphic property of Toom–Cook by caching the images and applying the Toom–Cook interpolations to the sums of the images. Later, [IKPC20] applied the Toeplitz-TC approach and [BMK+21] proposed striding Toom–Cook approach. Both approaches exploit the structure of the polynomial modulus and reduced the memory operations. The Toeplitz-TC approach remains the fastest approach computing over $\mathbb{Z}_{2^k}$ with 16-bit DSP instructions.

NTTs. [CHK+21] applied the 32-bit NTT approach, Michiel van Beirendonck optimized their stack usage, and [ACC+21] furthered the stack optimizations with 16-bit/32-bit NTTs over a product of two 16-bit NTT-friendly moduli. 32-bit NTT is the fastest approach for the matrix-vector multiplications and inner products for Saber on Cortex-M4. See Table 16.7 for a summary of the performance.

16.4.3 Scheme

For the overall performance of Saber on Cortex-M4, we summarize the performance of the Toeplitz-TC approach by [IKPC20] and the speed-optimized and stack-optimized implementations with 16-bit/32-bit NTTs by [ACC+21]. See Table 16.8 for a summary of the performance.

Table 16.8: Performance cycles of Saber on Cortex-M4.

Parameter set	Work	**K**	**E**	**D**
`lightsaber`	[IKPC20]*	376k	528k	547k
	[ACC+21]* (stack)	382k	534k	535k
	[ACC+21]* (speed)	**313k**	**424k**	**408k**
`saber`	[IKPC20]*	728k	940k	966k
	[ACC+21]* (stack)	741k	956k	953k
	[ACC+21]* (speed)	**569k**	**721k**	**693k**
`firesaber`	[IKPC20]*	1 166k	1 430k	1 479k
	[ACC+21]* (stack)	1 197k	1 464k	1 468k
	[ACC+21]* (speed)	**874k**	**1 057k**	**1 028k**

* Benchmark of this thesis.

16.5 Reviewing Armv8-A Neon Implementations

16.5.1 `NTT` and $\mathtt{NTT}^{-1}$

For the NTT approaches with Armv8-A Neon, [NG21] followed the AVX2 implementations by [CHK$^+$21] and computed the products over two 16-bit NTT-friendly moduli with 16-bit multiplication instructions. It is unclear what the motivation was – For the AVX2 implementation, [CHK$^+$21] implemented with 16-bit multiplication instructions due to the lack of 32-bit high multiplications for efficient vectorization, but we have 32-bit high multiplication instructions in Armv8-A Neon. [BHK$^+$21] implemented the 32-bit NTT approach and outperformed [NG21]. See Table 16.9 for a summary of the performance.

Table 16.9: `NTT` and $\mathtt{NTT}^{-1}$ for Saber with Armv8-A Neon on Cortex-A72 and Firestorm.

Operation	Work	Cortex-A72	Firestorm
`NTT`	[NG21]*	3 982	1 078
	[BHK$^+$21]**	**1 530/2 039**	**301/411**
$\mathtt{NTT}^{-1}$	[NG21]*	3 786	1 062
	[BHK$^+$21]**	**1 898**	**390**

*16-bit NTT approach. The numbers are scaled by two since two calls are required for implementing a polynomial multiplication in Saber.
**32-bit NTT approach. Benchmark of this thesis.

16.5.2 Matrix-Vector Multiplication and Inner Product

For the matrix-vector multiplications and inner products, [NG21] implemented Toom–Cook over $\mathbb{Z}_{2^k}$ and the 16-bit NTT approaches. They concluded that the Toom–Cook over $\mathbb{Z}_{2^k}$ was more preferable than the 16-bit NTT approach while taking the performance on Cortex-A72 and Apple M1 into account. After the deployment of 32-bit NTT and Barrett multiplication by [BHK$^+$21], the NTT approach was significantly faster than the Toom–Cook approach. See Table 16.10 for a summary of the performance.

Table 16.10: Performance cycles of matrix-vector multiplications, inner products in encryptions, and inner products in decryptions of Saber with Armv8-A Neon on Cortex-A72 and Firestorm.

Operation	Work	`lightsaber`	`saber`	`firesaber`
Cortex-A72				
MV	[NG21]**	39 106	76 253	129 887
	[BHK+21]*	**17 978**	**34 439**	**58 821**
IP(Enc)	[NG21]**	-	-	-
	[BHK+21]*	**6 943**	**9 468**	**11 884**
IP(Dec)	[NG21]**	16 610	22 529	28 364
	[BHK+21]*	**10 999**	**15 765**	**21 523**
Asymmetric mul.	[BHK+21]*	1 996	2 780	3 711
Firestorm				
MV	[NG21]**	6 568	12 970	21 336
	[BHK+21]*	**3 938**	**7 570**	**12 079**
IP(Enc)	[NG21]**	-	-	-
	[BHK+21]*	**1 558**	**2 092**	**2 612**
IP(Dec)	[NG21]**	3 181	4 248	5 323
	[BHK+21]*	**2 377**	**3 381**	**4 253**
Asymmetric mul.	[BHK+21]*	566	800	1 021

*Benchmark of this thesis.
**Toom–Cook approach. Benchmark of this thesis.

16.5.3 Scheme

For the overall performance of Saber on Cortex-A72 and Firestorm. the 32-bit NTT approach by [BHK+21] significantly outperformed the fastest implementations by [NG21], where the majority of the improvement comes from the significantly improved matrix-vector multiplications and inner products. See Table 16.11 for a summary of the performance.

Table 16.11: Performance cycles of Saber with Armv8-A Neon on Cortex-A72 and Firestorm.

Parameter set	Work	**K**	**E**	**D**
Cortex-A72				
`lightsaber`	[NG21]**	113 599	124 806	124 245
	[BHK+21]*	**94 033**	**96 555**	**89 668**
`saber`	[NG21]**	180 587	203 831	207 018
	[BHK+21]*	**140 704**	**151 511**	**148 497**
`firesaber`	[NG21]**	274 638	305 733	313 114
	[BHK+21]*	**203 281**	**224 104**	**220 031**
Firestorm				
`lightsaber`	[NG21]**	27 097	36 858	36 207
	[BHK+21]*	**22 753**	**31 815**	**30 426**
`saber`	[NG21]**	48 382	59 664	59 033
	[BHK+21]*	**37 925**	**49 276**	**47 778**
`firesaber`	[NG21]**	74 483	89 775	117 764
	[BHK+21]*	**58 169**	**72 283**	**70 329**

*Benchmark of this thesis.
**Toom–Cook approach. Benchmark of this thesis.

16.6 Reviewing AVX2 Implementations

For the AVX2-optimized implementations of Saber, this thesis reviews and benchmarks the implementations by [MKV20, CHK+21].

16.6.1 NTT and NTT^{-1}

For the AVX2-optimized 16-bit NTT approach for Saber, [CHK+21] computed `NTT`s and `NTT`$^{-1}$s over two 16-bit NTT-friendly moduli, and recovered the desired results over $\mathbb{Z}_{2^k}$ with the divided-difference form of Chinese remainder theorem [CHK+21, Theorem 1]. There are several choices for the pair of 16-bit NTT moduli, and [CHK+21] followed [BBC+20]'s AVX2-optimized NTT implementation. See Table 16.12 for a summary of the performance.

Table 16.12: Performance cycles of 16-bit NTT and NTT^{-1} for Saber with AVX2 on Haswell.

Work	NTT	NTT^{-1}
[CHK⁺21]*	328	288

*Benchmark of this thesis.

16.6.2 Matrix-Vector Multiplication and Inner Product

For the matrix-vector multiplications and inner products of Saber with AVX2, there are two approaches in the literature: the Toom–Cook by [MKV20] and the 16-bit NTT approach by [CHK⁺21]. Since the NTT approach nicely aligns with the structure of matrix-vector multiplication (cf. Section 16.2), [CHK⁺21]'s 16-bit NTT approach significantly outperformed prior Toom–Cook by [MKV20]. One should note that inner product in the decryption of [CHK⁺21] operates differently from the specification of Saber as the secret-key operand was stored in the NTT domain in the AVX2 implementation of [CHK⁺21]. For ensuring consistent testvectors, this thesis implements the AVX2-optimized inner products in decryptions in align with Saber specification. See Table 16.13 for a summary of the performance.

Table 16.13: Performance cycles of matrix-vector multiplications, inner products in encryptions, and inner products in decryptions of Saber with AVX2 on Haswell.

Operation	Work	`lightsaber`	`saber`	`firesaber`
MV	[MKV20]*	8 154	16 529	27 734
	[CHK⁺21]*	**6 244**	**11 945**	**19 222**
IP(Enc)	[MKV20]*	3 709	5 098	6 546
	[CHK⁺21]*	**2 435**	**3 283**	**4 103**
IP(Dec)	[MKV20]*	4 421	6 171	7 957
	[CHK⁺21]*	**3 811**	**5 378**	**6 878**
IP(NTT)	[CHK⁺21]*	178	240	307

*Benchmark of this thesis.

16.6.3 Scheme

For the overall performance of Saber, this thesis integrates the modified inner products in the decryptions and all other AVX2 implementations by [CHK+21] into the C implementations of Saber by [ACC+21]. See Table 16.14 for a summary of the performance.

Table 16.14: Performance cycles of Saber with AVX2 on Haswell.

Parameter set	Work	**K**	**E**	**D**
`lightsaber`	[MKV20]*	52 195	67 907	65 622
	[CHK+21]*	**49 994**	**64 682**	**61 745**
`saber`	[MKV20]*	92 838	114 006	110 308
	[CHK+21]*	**87 738**	**106 924**	**102 672**
`firesaber`	[MKV20]*	145 356	171 258	168 000
	[CHK+21]*	**136 504**	**159 812**	**155 330**

*Benchmark of this thesis.

Bibliography

[AAB$^+$25] José Bacelar Almeida, Gustavo Xavier Delerue Marinho Alves, Manuel Barbosa, Gilles Barthe, Luís Esquível, Vincent Hwang, Tiago Oliveira, Hugo Pacheco, Peter Schwabe, and Pierre-Yves Strub. Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3820–3838. IEEE, 2025. 13

[AAC$^+$22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. NISTIR 8413-upd1 – status report on the third round of the nist post-quantum cryptography standardization process, 2022. `https://doi.org/10.6028/NIST.IR.8413-upd1`. 2

[AASA$^+$19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. NISTIR 8240 – status report on the first round of the nist post-quantum cryptography standardization process, 2019. `https://doi.org/10.6028/NIST.IR.8240`. 2

[AASA$^+$20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. NISTIR 8309 – status report on the second round of the nist post-quantum cryptography standardization process, 2020. `https://doi.org/10.6028/NIST.IR.8309`. 2, 311

[AB74] Ramesh C. Agarwal and Charles S. Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974. `https://ieeexplore.ieee.org/abstract/document/1162555`. 48, 49, 359

[ABB$^+$17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017. `https://dl.acm.org/doi/10.1145/3133956.3134078`. 298, 314

[ABB$^+$20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020. 298

[ABB$^+$23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber Episode IV: Implementation Correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, 2023. `https://tches.iacr.org/index.php/TCHES/article/view/10960`. 298

[ABC$^+$25] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. NIST IR 8545 – status report on the fourth round of the nist post-quantum cryptography standardization process, 2025. `https://doi.org/10.6028/NIST.IR.8545`. 2

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R, M} LWE schemes.

IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3):336–357, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8593`. 136, 143, 206

[ABD$^+$20a] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. xxii, 147, 149, 183, 197, 198, 317, 352

[ABD$^+$20b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. 199, 203, 204, 205, 206, 207, 208, 211, 213, 214, 215, 216, 314, 317

[ABKK23] Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, and Fabien Klein. Fast and Clean: Auditable high-performance assembly via constraint solving. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):87–132, 2023. `https://tches.iacr.org/index.php/TCHES/article/view/11241`. 195, 196, 208, 209

[ABRB$^+$19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1622, 2019. `https://dl.acm.org/doi/10.1145/3319535.3363211`. 298, 299

[ACC$^+$20] Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime: Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2020. `https://tches.iacr.org/index.`

php/TCHES/article/view/8733. 8, 9, 65, 99, 134, 143, 193, 206, 209, 224, 236, 237, 317

[ACC⁺21] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127–151, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9292. 9, 17, 65, 165, 190, 191, 192, 206, 257, 258, 259, 260, 261, 262, 263, 268, 313, 346, 358

[Aff13] Reynald Affeldt. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, 9(2):59–77, 2013. 314

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, pages 853–871. Springer, 2022. https://link.springer.com/chapter/10.1007/978-3-031-09234-3_42. 9, 137, 143, 187, 190, 193, 194, 195, 206, 207, 208

[AHY22] Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):349–371, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9823. 17, 48, 74, 171, 180, 223, 224, 225, 237, 238, 241

[AM07] Reynald Affeldt and Nicolas Marti. An approach to formal verification of arithmetic functions in assembly. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science*, volume 4435 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2007. 314

[AMD25] AMD. *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0*, 2025. https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build. 111

[AMOT22] Daichi Aoki, Kazuhiko Minematsu, Toshihiko Okamura, and Tsuyoshi Takagi. Efficient Word Size Modular Multiplication over Signed Integers. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 94–101. IEEE, 2022. `https://ieeexplore.ieee.org/document/9974215`. 38, 41, 42, 140, 141, 142, 143, 203

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of BBS. *Science of Computer Programming*, 77(10–11):1058–1074, 2012. 314

[App15] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):7:1–7:31, 2015. 314

[Arm] Arm Ltd. Mbed TLS. `https://tls.mbed.org/`. 371, 372

[ARM10a] ARM. *Cortex-M3 Technical Reference Manual*, 2010. `https://developer.arm.com/documentation/ddi0337/h`. 116, 142, 155

[ARM10b] ARM. *Cortex-M4 Technical Reference Manual*, 2010. `https://developer.arm.com/documentation/ddi0439/b/`. 117, 143, 155

[ARM10c] ARM. *Cortex-M4 Technical Reference Manual ARM DDI 0439B Errata 01*, 2010. `https://developer.arm.com/documentation/ddi0439/latest/`. 117, 143, 155

[ARM15] ARM. *Cortex-A72 Software Optimization Guide*, 2015. `https://developer.arm.com/documentation/uan0016/a/`. 117, 118, 119

[ARM21a] ARM. *Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile*, 2021. `https://developer.arm.com/documentation/ddi0487/gb/?lang=en`. 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111

[ARM21b] ARM. *Armv7-M Architecture Refernce Manual*, 2021. `https://developer.arm.com/documentation/ddi0403/ed`. 91, 92, 93, 94, 95, 96, 97, 98, 99, 100

[ARM21c] ARM. *Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)*, 2021. `https://github.com/ARM-software/abi-aa/releases`. 100, 101

[ARM21d] ARM. *Procedure Call Standard for the Arm® Architecture*, 2021. `https://github.com/ARM-software/abi-aa/releases`. 91, 99

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *CRYPTO 1986*, LNCS, pages 311–323. SV, 1986. `https://link.springer.com/chapter/10.1007/3-540-47721-7_24`. 38

[BBC$^+$20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. xvii, 13, 65, 221, 228, 231, 235, 236, 247, 266, 313, 317, 352

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, 2022. `https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein`. 12, 67, 81, 82, 174, 175, 239, 240, 242, 243, 249, 250

[BBF$^+$21] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. Easypqc: Verifying post-quantum cryptography. Cryptology ePrint Archive, Report 2021/1253, 2021. `https://ia.cr/2021/1253`. 314

[BC87] J. V. Brawley and L. Carlitz. Irreducibles and the composed product for polynomials over a finite field. *Discrete Mathematics*, 65(2):115–139, 1987. `https://www.sciencedirect.com/science/article/pii/0012365X8790135X`. 51, 71

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In *Cryptographic Hardware and Embedded Systems-CHES 2013: 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings 15*, pages 250–272. Springer, 2013. `https://link.springer.com/chapter/10.1007/978-3-642-40349-1_15#:~:text=Abstract,a%20single%20Ivy%20Bridge%20core`. 77

[BCS16] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. *J. Cryptol.*, 29(4):775–805, 2016. 312, 313

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001. `https://cr.yp.to/papers.html#m3`. 14, 62, 63, 74

[Ber05] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *International workshop on fast software encryption*, pages 32–49. Springer, 2005. `https://doi.org/10.1007/11502760_3`. 298

[Ber07] Daniel J. Bernstein. The tangent FFT. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 17th International Symposium, AAECC-17*, pages 291–300, 2007. `https://link.springer.com/chapter/10.1007/978-3-540-77224-8_34`. 51

[Ber08a] Daniel J. Bernstein. Chacha, a variant of salsa20. In *Workshop record of The State of the Art of Stream Ciphers*, pages 273–278. Citeseer, 2008. `https://www.ecrypt.eu.org/stvl/sasc2008/SASCRecord.zip`. 180, 297

[Ber08b] Daniel J. Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008. `https://cr.yp.to/papers.html#multapps`. 14, 72

[Ber23] Daniel J. Bernstein. Fast norm computation in smooth-degree abelian number fields. *Research in Number Theory*, 9(4):82, 2023. `https://link.springer.com/article/10.1007/s40993-022-00402-0`. 73

[BGM93] Ian F. Blake, Shuhong Gao, and Ronald C. Mullin. Explicit Factorization of $x^{2^k}+1$ over $\mathbb{F}_p$ with Prime $p \equiv 3 \bmod 4$. *Applicable Algebra in Engineering, Communication and Computing*, 4(2):89–94, 1993. `https://link.springer.com/article/10.1007/BF01386832`. 51

[BHK$^+$21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/9295`. 9, 11, 17, 37, 38, 82, 144, 145, 148, 181, 195, 196, 197, 202, 206, 208, 209, 210, 213, 264, 265, 266, 360, 363

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number–Theoretic Transforms. In *International Workshop on Security*, pages 3–23. Springer, 2022. `https://link.springer.com/chapter/10.1007/978-3-031-15255-9_1`. 16, 38

[BKL+17] Daniel J Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, et al. Gimli: a cross-platform permutation. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 299–320. Springer, 2017. `https://link.springer.com/chapter/10.1007/978-3-319-66787-4_15`. 298

[BMGVdO15] F.E. Brochero Martínez, C. R. Giraldo Vergaraand, and L. Batista de Oliveira. Explicit factorization of $x^n - 1 \in \mathbb{F}_q[x]$. *Designs, Codes and Cryptography*, 77:277–286, 2015. `https://link.springer.com/article/10.1007/s10623-014-0005-y`. 51

[BMK+21] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhedeg. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, 2021. `https://tches.iacr.org/index.php/TCHES/article/view/9305`. 74, 254, 260, 261, 262, 363, 368, 369

[Bou89] Nicolas Bourbaki. *Algebra I*. Springer, 1989. 24

[BPYA15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *USENIX Security Symposium*, pages 207–221. USENIX Association, 2015. 314

[Bra84] Ronald N. Bracewell. The Fast Hartley Transform. *Proceedings of the IEEE*, 72(8):1010–1018, 1984. `https://ieeexplore.ieee.org/document/1457236`. 51

[Bru78] Georg Bruun. z-transform DFT Filters and FFT's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978. `https://ieeexplore.ieee.org/document/1163036`. 51

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/8298`. 220, 236

[CCHY24] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU. In Anupam Chattopadhyay, Shivam Bhasin, Stjepan Picek, and Chester Rebeiro, editors, *Progress in Cryptology – INDOCRYPT 2023*, pages 177–196. Springer, 2024. `https://link.springer.com/chapter/10.1007/978-3-031-56235-8_9`. 11, 82, 85, 86, 181, 225, 226, 227, 228, 229

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. 13, 65, 217, 227, 316, 317

[CF94] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-integer Arithmetic. *Mathematics of computation*, 62(205):305–324, 1994. `https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/?active=current`. 48, 49, 72

[Che21] Yun-Li Cheng. Number Theoretic Transform for Polynomial Multiplication in Lattice-based Cryptography on ARM Processors. Master's thesis, 2021. `https://github.com/dean3154/ntrup_m4`. 237, 238

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. `https://tches.iacr.org/index.php/`

TCHES/article/view/8791. 9, 17, 65, 67, 189, 193, 206, 209, 214, 220, 223, 224, 229, 230, 257, 260, 261, 262, 263, 264, 266, 267, 268, 317, 341, 347, 362

[CK91] David G. Cantor and Erich Kaltofen. On Fast Multiplication of Polynomials over Arbitrary Algebras. *Acta Informatica*, 28(7):693–701, 1991. https://link.springer.com/article/10.1007/BF01178683. 60

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software Analysis Perspective. In *Software Engineering and Formal Methods: 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings 10*, pages 233–247. Springer, 2012. 304

[CPS$^+$20] Chitchanok Chuengsatiansup, Thomas Prest, Damien Stehlé, Alexandre Wallet, and Keita Xagawa. ModFalcon: Compact signatures based on module-NTRU lattices. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 853–866, 2020. 310

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/. 49, 317

[Dan24] Daniel J. Bernstein and Karthikeyan Bhargavan and Shivam Bhasin and Anupam Chattopadhyay and Tee Kiah Chia and Matthias J. Kannwischer and Franziskus Kiefer and Thales Paiva and Prasanna Ravi and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024. 5

[DGPY20] Léo Ducas, Steven Galbraith, Thomas Prest, and Yang Yu. Integral Matrix Gram Root and Lattice Gaussian Sampling without Floats. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 608–637. Springer, 2020. 310

[DH84] Pierre Duhamel and Henk Hollmann. 'Split Radix' FFT Algorithm. *Electronics letters*, 20(1):14–16, 1984. `https://digital-library.theiet.org/content/journals/10.1049/el_19840012`. 51, 70

[Dhe03] Jean-François Dhem. Efficient Modular Reduction Algorithm in $\mathbb{F}_q[x]$ and Its Application to "Left to Right" Modular Multiplication in $\mathbb{F}_2[x]$. In *Cryptographic Hardware and Embedded Systems-CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings 5*, pages 203–213. Springer, 2003. `https://link.springer.com/chapter/10.1007/978-3-540-45238-6_17`. 38

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. 65, 251, 315, 317

[DP16] Léo Ducas and Thomas Prest. Fast Fourier Orthogonalization. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, pages 191–198, 2016. 295, 310

[DV78a] Eric Dubois and Anastasios N. Venetsanopoulos. A New Algorithm for the Radix-3 FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(3):222–225, 1978. `https://ieeexplore.ieee.org/document/1163084`. 241

[DV78b] Eric Dubois and Anastasios N. Venetsanopoulos. The discrete Fourier transform over finite rings with application to fast convolution. *IEEE Computer Architecture Letters*, 27(07):586–593, 1978. `https://ieeexplore.ieee.org/document/1675158`. 49

[DV90] Pierre Duhamel and Martin Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal processing*, 19(4):259–299, 1990. `https://www.sciencedirect.com/science/article/pii/016516849090158U`. 14

[EFG$^+$22] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. pages 222–253, 2022. 310

[EPG$^+$19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE, 2019. 314, 319

[FD05] Haining Fan and Yiqi Dai. Fast Bit-Parallel $GF(2^n)$ Multiplier for All Trinomials. *IEEE Transactions on Computers*, 54(4):485–490, 2005. `https://ieeexplore.ieee.org/document/1401867`. 79

[FH07] Haining Fan and M. Anwar Hasan. A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields. *IEEE Transactions on Computers*, 56(2):224–233, 2007. `https://ieeexplore.ieee.org/document/4042682`. 79

[Fid73] Charles M. Fiduccia. *On the Algebraic Complexity of Matrix Multiplication*. PhD thesis, Brown University, 1973. `https://cr.yp.to/bib/entries.html#1973/fiduccia-matrix`. 77

[Flo72] Robert W Floyd. Permuting Information in Idealized Two-Level Storage. In *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pages 105–109. Springer, 1972. `https://link.springer.com/chapter/10.1007/978-1-4684-2001-2_10`. 82

[FLS$^+$19] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed Cryptographic Program Verification with Typed Cryptoline. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1606, 2019. `https://dl.acm.org/doi/abs/10.1145/3319535.3354199`. 297, 314, 319

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO '99*, volume 1666, pages 537–554, 1999. `http://dx.doi.org/10.1007/3-540-48405-1_34`. 316

[Fog] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD* *and VIA CPUs*. `https://www.agner.org/optimize/instruction_tables.pdf`. 120

[Fre] Free Software Foundation. The GNU multiple precision arithmetic library. `https://gmplib.org/`. 355

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8683`. 65

[Für09] Martin Fürer. Faster Integer Multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. `https://doi.org/10.1137/070711761`. 49, 70, 317, 319, 354, 357

[Gar07] Luis Carlos Coronado García. Can Schönhage multiplication speed up the RSA decryption or encryption? MoraviaCrypt 2007, 2007. 355

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8725`. 134, 135, 142, 143, 188, 189, 190, 192, 209

[GKZ07] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage–Strassen's large integer multiplication algorithm. In *ISSAC 2007*, pages 167–174. ACM, 2007. 355

[Goo58] I. J. Good. The Interaction Algorithm and Practical Fourier Analysis. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):361–372, 1958. `https://www.jstor.org/stable/2983896`. 52

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996. 2

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: For Fun and Profit. In *Proceedings of the November 7-10, 1966, Fall*

Joint Computer Conference, AFIPS '66 (Fall), pages 563–578. Association for Computing Machinery, 1966. https://doi.org/10.1145/1464291.1464352. 70, 318

[Har42] Ralph VL Hartley. A More Symmetrical Fourier Analysis Applied to Transmission Problems. *Proceedings of the IRE*, 30(3):144–150, 1942. https://ieeexplore.ieee.org/document/1694454. 51

[Has22] Chenar Abdulla Hassan. Radix-3 NTT-Based Polynomial Multiplication for Lattice-Based Cryptography. Master's thesis, Middle East Technical University, 2022. https://open.metu.edu.tr/handle/11511/97928. 48

[HAZ$^+$24] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):1–24, 2024. https://tches.iacr.org/index.php/TCHES/article/view/11419. 188, 190, 191, 192, 193, 194, 195, 205

[HB95] M.A. Hasan and V.K. Bhargava. Architecture for a low complexity rate-adaptive Reed-Solomon encoder. *IEEE Transactions on Computers*, 44(7):938–942, 1995. https://ieeexplore.ieee.org/document/392853. 79

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography*, volume 10677, pages 341–371, 2017. https://eprint.iacr.org/2017/604. 314, 315

[HKS23] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Barrett Multiplication for Dilithium on Embedded Devices. Cryptology ePrint Archive, Paper 2023/1955, 2023. https://eprint.iacr.org/2023/1955. 10, 32, 35, 36, 37, 38, 139

[HKS24] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying Polynomials without Powerful Multiplication Instructions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):160–202, 2024. https://tches.iacr.org/index.php/TCHES/article/view/11926. Extended from https://

eprint.iacr.org/2023/1955. Full version available at https://eprint.iacr.org/2024/1649. 10, 142, 143, 188, 189, 190, 191, 192, 193, 256, 257, 258, 259, 260

[HLS$^+$22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 718–750, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9838. 16, 297

[HLY24] Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. In *International Conference on Applied Cryptography and Network Security*, pages 24–46. Springer, 2024. https://link.springer.com/chapter/10.1007/978-3-031-54773-7_2. 12, 81, 82, 174, 239, 242, 245, 246, 247, 248

[HMCS77] David B. Harris, James H. McClellan, David S. K. Chan, and Hans W. Schuessler. Vector Radix Fast Fourier Transform. In *ICASSP'77. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 548–551, 1977. https://ieeexplore.ieee.org/document/1170349. 54

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory – ANTS-III*, pages 267–288, 1998. http://dx.doi.org/10.1007/BFb0054868. 316

[HQZ04] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The Middle Product Algorithm I. *Applicable Algebra in Engineering, Communication and Computing*, 14(6):415–438, 2004. https://link.springer.com/article/10.1007/s00200-003-0144-2. 77

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(nlogn)$. *Annals of Mathematics*, 193(2):563–617, 2021. https://annals.math.princeton.edu/2021/193-2/p04. 317, 319, 354, 356, 357

[HVDH22] David Harvey and Joris Van Der Hoeven. Polynomial Multiplication over Finite Fields in time $O(nlogn)$. *Journal of the ACM*, 69(2):1–40, 2022. https://dl.acm.org/doi/full/10.1145/3505584. 55

[Hwa22] Vincent Hwang. Case Studies on Implementing Number–Theoretic Transforms with Armv7-M, Armv7E-M, and Armv8-A. Master's thesis, 2022. https://github.com/vincentvbh/NTTs_with_Armv7-M_Armv7E-M_Armv8-A. 14, 72, 92, 191, 255

[Hwa24a] Vincent Hwang. A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems. *IACR Communications in Cryptology*, 1(2), 2024. https://cic.iacr.org/p/1/2/1. 14, 39, 41

[Hwa24b] Vincent Hwang. Formal verification of emulated floating-point arithmetic in falcon. In *International Workshop on Security*, pages 125–141. Springer, 2024. 15

[Hwa24c] Vincent Hwang. Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime. In *Australasian Conference on Information Security and Privacy*, pages 84–102. Springer, 2024. https://link.springer.com/chapter/10.1007/978-981-97-5028-3_5. 12, 81, 82, 84, 243, 245, 246, 247, 248, 250

[HZZ$^+$22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833. 38, 41, 42, 140, 141, 143, 206, 207, 208

[HZZ$^+$24] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Cetin Kaya Koc, and Donglong Chen. Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices. *IEEE Transactions on Information Forensics and Security*, 2024. https://ieeexplore.ieee.org/document/10453274. 140, 142, 203, 204

[IKPC20] Írem Keskinkurt Paksoy and Murat Cenk. TMVP-based Multiplication for Polynomial Quotient Rings and Application to Saber on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2020/1302, 2020. https://eprint.iacr.org/2020/1302. 224, 254, 256, 260, 261, 262, 263

[IKPC22] Írem Keskinkurt Paksoy and Murat Cenk. Faster NTRU on ARM Cortex-M4 with TMVP-based multiplication.

IEEE Transactions on Circuits and Systems I: Regular Papers, 69(10):4083–4092, 2022. `https://ieeexplore.ieee.org/document/9835023`. 80, 223, 224, 225

[Jac12a] Nathan Jacobson. *Basic Algebra I*. Courier Corporation, 2012. 21, 24

[Jac12b] Nathan Jacobson. *Basic Algebra II*. Courier Corporation, 2012. 21

[JF07] Steven G. Johnson and Matteo Frigo. A Modified Split-Radix FFT With Fewer Arithmetic Operations. *IEEE Transactions on Signal Processing*, 55(1):111–119, 2007. `https://ieeexplore.ieee.org/document/4034175`. 51

[Joh] Dougall Johnson. *SIMD and FP Instructions (Firestorm)*. `https://dougallj.github.io/applecpu/firestorm-simd.html`. 119

[JWYC24] Li-Jie Jian, Ting-Yuan Wang, Bo-Yin Yang, and Ming-Shing Chen. Jumping for Bernstein-Yang Inversion. Cryptology ePrint Archive, Paper 2024/644, 2024. 220, 247

[KAK96] Cetin Kaya Koc, Tolga Acar, and Burton S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996. `https://ieeexplore.ieee.org/document/502403`. 361

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145(2), pages 293–294, 1962. `http://cr.yp.to/bib/1963/karatsuba.html`. 45, 69, 354

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates. In *International Conference on Applied Cryptography and Network Security*, pages 281–301. Springer, 2019. `https://link.springer.com/chapter/10.1007/978-3-030-21568-2_14`. 172, 223, 224, 253, 260, 261, 262

[Li21] Ching-Lin Li. Implementation of Polynomial Modular Inversion in Lattice-based cryptography on ARM. Master's thesis, 2021. `https://github.com/trista5658321/polyinv-m4`. 220, 223

[LLS$^+$23] Li-Chang Lai, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Automatic Verification of Cryptographic Block Function Implementations with Logical Equivalence Checking. Cryptology ePrint Archive, Paper 2023/1861, 2023. `https://eprint.iacr.org/2023/1861`. 297

[LLZ$^+$18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, and Bao Li. LAC: practical ring-lwe based public-key encryption with byte-level modulus. *IACR Cryptol. ePrint Arch.*, 2018. `https://eprint.iacr.org/2018/1009`. 313, 352

[LST$^+$19] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic in Cryptographic C Programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 552–564. IEEE, 2019. `https://ieeexplore.ieee.org/document/8952256`. 297, 314, 319

[LVB07] T. Lundy and James Van Buskirk. A new matrix approach to real FFTs and convolutions of length 2^k. *Computing*, 80:23–45, 2007. `https://link.springer.com/article/10.1007/s00607-007-0222-6`. 51

[LZ22] Zhichuang Liang and Yunlei Zhao. Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey. *arXiv preprint arXiv:2211.13546*, 2022. `https://arxiv.org/abs/2211.13546`. 14

[MC13] Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2013. 314

[Mey96] Helmut Meyn. Factorization of the Cyclotomic Polynomial $x^{2^n} + 1$ over Finite Fields. *Finite Fields and Their Applications*, 2(4):439–442, 1996. `https://www.sciencedirect.com/science/article/pii/S107157979690026X`. 51

[MG07] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2007. 314

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8550`. 253, 261, 262, 266, 267, 268

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of computation*, 44(170):519–521, 1985. `https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/?active=current`. 35, 36, 361

[Mur96] Hideo Murakami. Real-valued fast discrete Fourier transform and cyclic convolution algorithms of highly composite even length. In *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, volume 3, pages 1311–1314, 1996. `https://ieeexplore.ieee.org/document/543667`. 51

[MV83a] Jean-Bernard Martens and Marc C. Vanwormhoudt. Convolution Using a Conjugate Symmetry Property for Number Theoretic Transforms Over Rings of Regular Integers. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1121–1125, 1983. `https://ieeexplore.ieee.org/document/1164198`. 62

[MV83b] Jean-Bernard Martens and Marc C. Vanwormhoudt. Convolutions of Long Integer Sequences by Means of Number Theoretic Transforms Over Residue Class Polynomial Rings. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1125–1134, 1983. `https://ieeexplore.ieee.org/abstract/document/1164201`. 62, 73

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists. In *Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings*, pages 234–254. Springer, 2021. `https://link.springer.com/chapter/10.1007/978-3-030-81293-5_13`. 82, 172, 181, 196, 208, 209, 210, 225, 226, 227, 228, 229, 264, 265, 266

[Nic71] Peter J. Nicholson. Algebraic Theory of Finite Fourier Transforms. *Journal of Computer and System Sciences*, 5(5):524–547, 1971. `https://www.sciencedirect.com/science/article/pii/S0022000071800144`. 65

[NIS] NIST, the US National Institute of Standards and Technology. Post-Quantum Cryptography Standardization Project. 293, 311

[Nus80] Henri J. Nussbaumer. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980. `https://ieeexplore.ieee.org/document/1163372`. 61, 63

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer Berlin, Heidelberg, 2nd edition, 1982. `https://doi.org/10.1007/978-3-642-81897-4`. 14

[PAA$^+$20] Thomas Pöppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. 313, 352

[PBT$^+$24] David Du Pont, Jonas Bertels, Furkan Turan, Michiel Van Beirendonck, and Ingrid Verbauwhede. Hardware Acceleration of the Prime-Factor and Rader NTT for BGV Fully Homomorphic Encryption. In *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*, pages 1–8. IEEE Computer Society, 2024. 170

[PFH$^+$20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. 15, 293, 294, 295, 317, 352

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021. `https://ieeexplore.ieee.org/document/9416314`. 38, 39, 41, 42

[Pol71] John M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of computation*, 25(114):365–374, 1971. `https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/?active=current`. 49, 65, 356

[Por] Thomas Pornin. BearSSL: a smaller TLS/SSL library. `https://bearssl.org/`. 371, 372

[Por19] Thomas Pornin. New Efficient, Constant-Time Implementations of Falcon, 2019. 293, 294, 296, 297, 306, 307, 308, 310

[Por23] Thomas Pornin. Improved Key Pair Generation for Falcon, BAT and Hawk, 2023. 309

[PQC21] PQClean. The PQClean project, 2021. 332

[PT18] Paul Pollack and Enrique Treviño. Finding the Four Squares in Lagrange's Theorem. *Integers*, 18A:A15, 2018. 310

[PTWY18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk). In *29th International Conference on Concurrency Theory (CONCUR 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2018.4`. 297, 314, 319

[Rad68] Charles M. Rader. Discrete Fourier Transforms When the Number of Data Samples Is Prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968. `https://ieeexplore.ieee.org/document/1448407`. 54

[RSA78] Ron Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 356

[Sch77] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977. `https://link.springer.com/article/10.1007/bf00289470`. 60, 61, 62

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Report 2018/039, 2018. `https://eprint.iacr.org/2018/039`. 35, 36, 143, 146, 148, 149, 213, 316, 317

[Sho] Victor Shoup. NTL: a Library for Doing Number Theory. `http://www.shoup.net/ntl/`. 37, 38

[Sho97a] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. 2

[Sho97b] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. 311

[Sho99] Victor Shoup. Efficient Computation of Minimal Polynomials in Algebraic Extensions of Finite Fields. In *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 53–58, 1999. `https://dl.acm.org/doi/10.1145/309831.309859`. 77

[SKS$^+$21] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Kyber on ARM64: Compact Implementations of Kyber on 64-bit ARM Cortex-A Processors. In *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II*, pages 424–440. Springer, 2021. `https://link.springer.com/chapter/10.1007/978-3-030-90022-9_23`. 144

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. `https://link.springer.com/article/10.1007/BF02242355`. 48, 49, 317, 354, 356

[STM20] STMicroelectronics. *STM32F205xx, STM32F207xx, Arm®-based 32-bit MCU, 150 DMIPs, up to 1 MB Flash/128+4KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 15 comm. interfaces and camera*, 2020. `https://www.st.com/resource/en/datasheet/stm32f207zg.pdf`. 179

[SXY18] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 520–551. Springer, 2018. 316

[Too63] Andrei L. Toom. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. `http://toomandre.com/my-articles/engmat/MULT-E.PDF`. 45, 69, 354

[TW13] Aleksandr Tuxanidy and Qiang Wang. Composed products and factors of cyclotomic polynomials over finite fields. *Designs, codes and cryptography*, 69(2):203–231, 2013. `https://link.springer.com/article/10.1007/s10623-012-9647-9`. 51

[TWY17] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1973–1987, 2017. `https://dl.acm.org/doi/abs/10.1145/3133956.3134076`. 297, 314, 319

[vdH04] Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, 2004. `https://dl.acm.org/doi/10.1145/1005285.1005327`. 72

[War12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley, 2012. 82

[Win80] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33. Society for Industrial and Applied Mathematics, 1980. `https://epubs.siam.org/doi/10.1137/1.9781611970364`. 45, 69, 76

[WY21] Yansheng Wu and Qin Yue. Further factorization of $x^n - 1$ over a finite field (II). *Discrete Mathematics, Algorithms and Applications*, 13(06):2150070, 2021. `https://www.worldscientific.com/doi/10.1142/S1793830921500701`. 51

[WYF18] Yansheng Wu, Qin Yue, and Shuqin Fan. Further factorization of $x^n - 1$ over a finite field. *Finite Fields and Their Applications*, 54:197–215, 2018. `https://www.sciencedirect.com/science/article/pii/S1071579718300996`. 51

[Yan23] Bo-Yin Yang, 2023. Personal communication. 79

[YGS$^+$17] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2020. ACM, 2017. 314

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806. ACM, 2017. 313

[ZCH$^+$19] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, and Oussama Danba. NTRU. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`. 229, 230

Appendix A

On Formal Verification

A.1 Paper: Formal Verification of Emulated Floating-Point Arithmetic in Falcon

We show that there is a discrepancy between the emulated floating-point multiplication in the submission package of the digital signature Falcon [PFH+20] and the claimed behavior. In particular, we show that some floating-point products with absolute values the smallest normal positive floating-point number are incorrectly zeroized. However, we show that the discrepancy does not affect the complex fast Fourier transforms in the signature generation of Falcon by modeling the floating-point addition, subtraction, and multiplication in CryptoLine. We later implement our own floating-point multiplications in Armv7-M assembly and Jasmin and prove their equivalence with our model, demonstrating the possibility of transferring the challenging verification task (verifying highly-optimized assembly) to the presumably more readable code base (Jasmin).

A.1.1 Introduction

Falcon [PFH+20] is one of the recently selected digital signatures for standardization by NIST [NIS]. Essentially, the signature is sampled with a probability approximated by floating-point arithmetic. Since floating-point arithmetic is not always constant-time, [Por19] implemented a series of constant-time floating-point arithmetic with software emulation. We show that

- the emulated floating-point multiplication does not honor its behavior claimed by [Por19];

- the discrepancy does not affect the complex fast Fourier transforms in the signature generation of Falcon; and
- how to prove the equivalence between emulated floating-point addition/-subtraction/multiplication implementations.

Our source code is publicly available at
`https://github.com/vincentvbh/Float_formal`.

A.1.2 Preliminaries

A.1.2.1 Falcon

Falcon is a lattice-based hash-and-sign digital signature based on fast Fourier sampling over an NTRU lattice [PFH+20]. The NTRU lattice is determined by four integer polynomials f, g, F, G satisfying

$$fG - gF = q \bmod (x^n + 1)$$

where $q = 12289$ and $n = 512, 1024$. The lattice is generated by the basis $\mathbf{B} = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$.

For the key generation, the four polynomials f, g, F, G form the secret key `sk` and hence must have small coefficients, and the public key `pk` is the polynomial $h = gf^{-1} \bmod (x^n + 1, q)$. See Algorithm A.1 for an illustration.

Algorithm A.1 Falcon key generation from the reference implementation.

Outputs: a public key `pk` and a secret key `sk`

```
(f, g) = mkgauss()   ▷ Generate $f, g$ from a discrete Gaussian distribution.
(F, G) = solve_NTRU($f, g, x^n + 1, q$)   ▷ $fG - gF = q \bmod (x^n + 1)$.
$h = gf^{-1} \bmod (x^n + 1, q)$
sk = $(f, g, F, G)$
pk = $h$
return pk, sk
```

For the signature generation, we generate a nonce r and hash it with the message m. We then start sampling two small polynomials s_1 and s_2 satisfying $s_1 + s_2 h = c \bmod (x^n + 1, q)$ where c is the hash. The signature is defined as (r, s_2). Falcon adopts the so-called fast Fourier sampling based on a randomized variant of fast Fourier nearest plane [DP16, PFH+20]. The idea essentially goes as follows: We compute $\hat{\mathbf{B}} = \texttt{FFT}\left(\mathbf{B}\right)$ and $\hat{c} = \texttt{FFT}\left(c\right)$ with complex fast Fourier transforms, compute $\mathbf{t} = \left(-\frac{\hat{c}\hat{F}}{q}, \frac{\hat{c}\hat{f}}{q}\right)$, construct the corresponding Falcon tree $\mathbf{T}$ from the LDL decomposition of $\hat{\mathbf{B}}\hat{\mathbf{B}}^*$, and apply fast Fourier nearest plane where the nearest plane part at the leaf level is replaced by a discrete Gaussian sampling with secret center constructed serially from $\mathbf{t}$ and prior samples and secret deviation constructed from $\mathbf{T}$. We refer to Algorithm A.2 for an overview of the signature generation and [PFH+20, Algorithim 11] for a more detailed explanation of the fast Fourier sampling.

Algorithm A.2 Falcon signature generation from the reference implementation.

Inputs: A message m and a secret key `sk`.
Outputs: A signature `sig`.

```
$r \leftarrow \{0,1\}^{320}$ uniformly                                ▷ Salt.
$c = \texttt{HashToPoint}\left(r||m\right)$
$\hat{c} = \texttt{FFT}(c)$
$\mathbf{B} = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$
$\hat{\mathbf{B}} = \begin{pmatrix} \hat{g} & -\hat{f} \\ \hat{G} & -\hat{F} \end{pmatrix} = \texttt{FFT}(\mathbf{B})$
$\mathbf{T} = \texttt{ffLDL}^*\left(\hat{\mathbf{B}}\hat{\mathbf{B}}^*\right)$
$\mathbf{T} = \texttt{Normalize}\left(\mathbf{T}\right)$
$\mathbf{t} = \left(\frac{\hat{-}c\hat{F}}{q}, \frac{\hat{c}\hat{f}}{q}\right)$                  ▷ $\mathbf{t} = \left(\hat{c}, 0\right)\hat{\mathbf{B}}^{-1}$
do
    do
        $\mathbf{z} = \texttt{ffSampling}\left(\mathbf{t}, \mathbf{T}\right)$
        $\mathbf{s} = \left(\mathbf{t} - \mathbf{z}\right)\hat{\mathbf{B}}$
    while $||\mathbf{s}||^2 > \lfloor \beta^2 \rfloor$
    $(s_1, s_2) = \texttt{iFFT}\left(\mathbf{s}\right)$
    $s = \texttt{Compress}\left(s_2, 8 \cdot \texttt{sbytelen} - 328\right)$
while $s == \perp$
$\texttt{sig} = (r, s)$
return sig
```

For the signature verification, we compute $s_1 = c - s_2 h \bmod (x^n + 1, q)$ and accept the signature if $||\left(s_1, s_2\right)||^2$ is small enough (reject otherwise). See

Algorithm A.3 for an illustration.

Algorithm A.3 Falcon signature verification.

Inputs: a message m, a signature `sig`, and a public key `pk` $= h$.

```
c = HashToPoint (r||m)
s_2 = Decompress (s, 8 · sbytelen − 328)
if s_2 == ⊥ then
    reject
s_1 = c − s_2 h
if ||(s_1, s_2)||^2 > ⌊β^2⌋ then
    reject
accept
```

A.1.2.2 Fast Fourier Transform

Fast Fourier transform (FFT) is a popular approach in signal processing, polynomial multiplication, and sampling. For a power of two n and the primitive $2n$th root of unity $\omega_{2n} \in \mathbb{C}$, the negacyclic Cooley–Tukey FFT transforms the polynomial ring $\mathbb{C}[x]/\langle x^n + 1\rangle$ into $\prod_{i=0,\ldots,n-1} \mathbb{C}[x]/\left\langle x - \omega_{2n}^{1+2i}\right\rangle$ in $O(n \log_2 n)$ operations in $\mathbb{C}$ up to the bit-reversal permutation. In Falcon, since the input coefficients are integers, [Por19] implemented an optimized variant of the complex Cooley–Tukey FFT with $\mathbb{C} \cong \mathbb{R}[z]/\left\langle z^2 + 1\right\rangle$. They also approximated the real number arithmetic by floating-point arithmetic in the signature generation.

A.1.2.3 Emulated Floating-Point Arithmetic

In Falcon, the real arithmetic in the signature generation is implemented as floating-point arithmetic. We briefly review the IEEE 754 double-precision floating-point specification.

A double-precision floating-point number is a 64-bit element consists of three parts (most significant bits first): a 1-bit `s` for the sign, an 11-bit `e` for the biased exponent, and a 52-bit `m` for the mantissa. We denote a floating-point as `s|e|m` with the sign `s`, the biased exponent `e`, and the mantissa `m`. When the biased exponent satisfies $0 < \mathtt{e} < 2047$, the floating-point number corresponds to the following real number:

$$(-1)^{\mathtt{s}} \, 2^{\mathtt{e}-1075} \left(2^{52} + \mathtt{m}\right).$$

We call such a floating-point number normal. In addition to the normal values, we also have the following special values:

- $\mathtt{e} = 0, \mathtt{m} = 0$: This corresponds to a zero value. Notice that there are two zeros ± 0 distinguished by the sign $\mathtt{s}$.
- $\mathtt{e} = 0, \mathtt{m} \neq 0$: This corresponds to the denormalized number $(-1)^{\mathtt{s}}\, 2^{\mathtt{e}-1074}\mathtt{m}$.
- $\mathtt{e} = 2047, \mathtt{m} = 0$: This corresponds to an infinity. Notice that there are also two infinities $\pm\infty$ distinguished by the sign $\mathtt{s}$.
- $\mathtt{e} = 2047, \mathtt{m} \neq 0$: This corresponds to a NaN (not-a-number) value.

In IEEE 754, "rounding to the nearest even" is adopted by default for rounding the real number result to a floating-point number. In Falcon, the authors claimed that infinities, NaNs, and denormalized numbers are not used and implemented a set of functions emulating the elementary floating-point arithmetic where the results are, according to their claim, correctly rounded for all normal values and zeros with "rounding to the nearest even" rule [Por19, Section 3.3]. We show that the latter does not hold, but it does not affect the complex fast Fourier transforms in the signature generation of Falcon.

A.1.2.4 CryptoLine

CryptoLine is a domain-specific language for modeling straight-line cryptographic programs. It was introduced by [TWY17, PTWY18] for verifying elliptic-curve arithmetic with assembly programs optimized "in the wild." In other words, assembly-optimized programs were first delivered by experts in assembly programming without considerations on verification, and verification effort was later devoted to verifying the resulting programs. CryptoLine was extended by [LST+19] for verifying elliptic-curve C implementations, and by [FLS+19] for signed arithmetic. Recently, [HLS+22] extended CryptoLine with compositional reasoning for verifying large dimensional number-theoretic transforms, and [LLS+23] extended CryptoLine with logical equivalence checking for the stream cipher ChaCha20 [Ber08a] and the cryptographic hash functions SHA-2 and SHA-3.

In CryptoLine, there are various instructions implementing the basic arithmetic, including signed/unsigned addition/subtraction/multiplication/extension, logical/arithmetic shift, bit-wise or/exclusive-or/and/not, bit-field splitting/concatenation, and conditional move. These instructions effectively capture the commonly used assembly instructions in cryptographic programs. We translate the target assembly programs into strings of CryptoLine instructions, and argue the properties of the strings of CryptoLine instructions.

There are two classes of predicates in CryptoLine for modeling the properties of strings of CryptoLine instructions: the algebraic predicates and the range predicates. An algebraic predicate is a conjunction of equations and modular equations, and a range predicate is a boolean formula with comparisons, equations, and modular equations. We have the assertion `assert` and the assumption `assume` annotations for imposing properties on the predicates. For an algebraic predicate `P` and a range predicate `Q`, `assert P && Q` asks the backend to verify `P` with the associated computer algebra system and `Q` with the associated SMT solver, and `assume P && Q` adds `P` and `Q` to the corresponding backend tools.

Assertions are used alone for verifying properties, and assumptions are commonly used in conjunction with assertions for transferring predicates between the backend tools. For example, we first verify an algebraic predicate `P` by imposing `assert P && true` and pass it to the SMT solver by imposing `assume true && P`.

For verifying a program as a whole, we specify pre-conditions on the variables, insert the string of CryptoLine instructions translated from the target program, annotate it with assertions and assumptions at proper locations, and finally specify the post-conditions. The most difficult part is the insertions of annotations, which, if ignored, results in non-responsiveness of the verification process in our context.

A.1.2.5 Jasmin

Jasmin is a programming language serving as a vehicle correlating assembly programs and their high-level abstractions. It was introduced by [ABB+17] for verifying the memory safety and constant-timeness of elliptic-curve arithmetic implementations. Jasmin was extended by [ABRB+19] for verifying implementation correctness and the security of SHA-3 implementations with EasyCrypt, and [ABB+20] revisited the compiler, memory model, and EasyCrypt embedding for verifying the ChaCha20 stream cipher, the Poly1305 message-authentication code [Ber05] , and the Gimli permutation [BKL+17]. Recently, [ABB+23] extended Jasmin with function calls, pointers to the stack memory, and the system call `randombytes`, and proved the implementation correctness of the key encapsulation mechanism Kyber recently selected by NIST as one of the to-be-standardized algorithms for post-quantum cryptography.

Programmers write Jasmin programs with similar control of the computational flows as in assembly, and compile the programs into assembly programs with the certified compiler `jasminc`. For verification, we extract the Jasmin programs to EasyCrypt according to the Jasmin model in EasyCrypt, and verify the desired properties with EasyCrypt. Compared to CryptoLine, verification in EasyCrypt requires much more effort by explicitly applying various lemmas instead of imposing properties in a declarative fashion in CryptoLine, but one can argue more properties in EasyCrypt, for example, the indifferentiability of SHA-3 from random oracle as shown in [ABRB+19].

A.1.3 Incorrect Zeroization

A.1.3.1 The Problem of Floating-Point Multiplication

We point out an incorrect zeroization in the emulated floating-point multiplications in Falcon. We illustrate the issue in the C reference implementation, and our finding also applies to the Armv7-M assembly-optimized implementation.

We briefly review the C reference implementation of the emulated floating-point multiplication in the submission package of Falcon as follows:

1. The inputs are two 64-bit integers with each representing a double-precision floating-point number.
2. Extract the mantissas and add them with 2^{52} as if the floating-point inputs are non-zero.
3. Compute the product of mantissas with radix-2^{25} arithmetic.
4. Normalize the product to a 55-bit value.
5. Compute the exponent field as the sum of input exponent fields with a corrective subtraction.
6. Compute the sign field as the exclusive-or of the input sign fields.
7. Zeroize the product if any of the input exponent fields is zero.
8. Zeroize the product if the resulting exponent is too small.
9. Zeroize the exponent field if the product is zero.
10. Assemble the sign field, exponent field, and the upper 53 bits of the 55-bit product.
11. Increment the resulting floating-point as an unsigned 64-bit integer if the 55-bit product should be rounded.

Algorithm A.4 Emulated C implementation (with some high-level syntax for the irrelevant parts for readability) of floating-point multiplication in Falcon. `fpr` is defined as `uint64_t`.

Input: `fpr` **elements** `x`, `y`.
Output: `fpr` **element** `z`.

```
uint64_t xu, yu, zu, z;
uint32_t z0, z1, sticky, round;
int ex, ey, e, d, s;
xu = $2^{52}$ | x & ($2^{52} - 1$);
yu = $2^{52}$ | y & ($2^{52} - 1$);
z0 + z1 * $2^{25}$ + zu * $2^{50}$ = xu * yu;
sticky = ((z0 | z1) + $2^{25} - 1$ ) » 25;   ▷ sticky $= 0$ if z0 $=$ z1 $= 0$, otherwise 1.
zu = zu | (uint64_t)sticky;
ex = (int)((x » 52) & ($2^{11} - 1$));
ey = (int)((y » 52) & ($2^{11} - 1$));
e = ex + ey - 2100;
(zu, e) $=$ normalize(zu, e, 55, sticky);
s = (int)((x ˆ y) » 63);
d = ((ex + $2^{11} - 1$) & (ey + $2^{11} - 1$)) » 11;   ▷ d $= 0$ if ex $= 0$ or ey $= 0$, otherwise 1.
zu = zu & (uint64_t)-d;   ▷ zu $= 0$ if d $= 0$, otherwise unchanged.
m = zu & ( ((uint32_t)(e + 1076) » 31) - 1);   ▷ m $= 0$ if e $< -1076$, otherwise unchanged.
e = e + 1076;
e = e & -((int)(uint32_t)(m » 54) );   ▷ e $= 0$ if $m = 0$, otherwise unchanged.
z = ( ((uint64_t)s « 63) | (m » 2) ) + ( (uint64_t)(uint32_t) e ) « 52;
round = (0xc8 » ((uint32_t)m & 7) ) & 1;   ▷ round $= 1$ if m & 7 $= 3, 6, 7$, otherwise 0.
z = z + (uint64_t)round;
return (fpr)z;
```

The issue is that the zeroization due to the smallness of the sum of the exponent fields should be the last operation since the increment as an unsigned 64-bit integer from rounding may results in an exponent field that is slightly above the zeroization threshold. We refer to Algorithm A.4 for a more detailed illustration where the line in red (blue) corresponds to the line in red (blue) of the above.

A.1.3.2 Extracting Witnesses

We show how to find inputs triggering an incorrect zeroization. For a floating-point number with exponent field e and mantissa m, if $1 \leq e \leq 1022$, $1 \leq m \leq 2^{52} - 2$, and $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor (2^{52} + m) \geq 2^{105} - 2^{51}$, then a floating-point with exponent field $1023 - e$ and mantissa $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor$ leads to an incorrect zeroization in Algorithm A.4 where the correct result is a floating-point number with absolute value the smallest normal positive floating-point number.

Recall that the issue of Algorithm A.4 is that the product is zeroized due to the smallness of the sum of exponents prior to the rounding at the end. We seek for conditions triggering both lines (if-conditions are taken) while the floating-point product is large enough after the rounding.

For simplicity, we first assume that the product of mantissas is an unsigned 105-bit integer (we will explain how this condition is satisfied shortly) so Line 12 changes nothing. We then choose `e` as the largest value, -1077, triggering Line 16 in Algorithm A.4:

```
m = zu & ( ((uint32_t)(e + 1076) ≫ 31) - 1).
```

This leads to the exponent fields `ex` $= e$ and `ey` $= 1023 - e$ after tracing the code (cf. Line 11). It remains to choose mantissas with a 105-bit product triggering Line 20:

```
round = (0xc8 ≫ ((uint32_t)m & 7) ) & 1.
```

This leads to the mantissas `xu` $= 2^{52} + m$ and `yu` $= \left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor$ with an m satisfying

- $1 \leq m \leq 2^{52} - 2$, and
- $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor (2^{52} + m) \geq 2^{105} - 2^{51}$.

This implies that we have $2^{55} - 2$ or $2^{55} - 1$ after normalizing to a 55-bit value (cf. Line 12), whose rounded value is 2^{55} if we round it prior to the zeroization in Line 16. Since the correct mantissa is 2^{55}, we have to increment the exponent by 1, removing the need of zeroization from the smallness of sum of the exponents.

Listing A.2 is our program testing if we can find a floating-point number `b` from the input floating-point number `a` whose floating-point product leads to an incorrect zeroization in Algorithm A.4, and Listing A.1 is an auxiliary function.

Listing A.1: Our C program testing if the input is small enough. We return 1 if `x` is small enough, and 0 otherwise.

```
int test_smallness(fpr x){

    fpr e = (x >> 52) & 0x7ff;
    fpr m = x & 0xfffffffffffff;

    if( (1 <= e) && (e <= 1022) )
        if( (1 <= m) && (m <= 0xffffffffffffe) )
            return 1;

    return 0;

}
```

Listing A.2: Our C program testing if there is an input leading to incorrect zeroization. If we find a floating-point value such that its floating-point product with `a` leads to incorrect zeroization, the floating-point value is stored in `*b` and 1 is returned. Otherwise, -1 is returned.

```
int retrieve_zeroization(fpr *b, fpr a){

    uint64_t t;

    __uint128_t a128, b128, t128;

    if(test_smallness(a) == 0)
        return -1;

    a128 = (1ULL << 52) + (a & 0xfffffffffffff);
    t128 = 1; t128 <<= 105;
    b128 = t128 / a128;

    if( a128 * b128 + (1ULL << 51) < t128)
        return -1;

    t = ( 1023 - ((a >> 52) & 0x7ff) ) << 52;
    t |= b128 - (1ULL << 52);
    *b = t;

    return 1;

}
```

A.1.3.3 An Example in Falcon

In Falcon, we need to approximate the real number $\frac{1}{\sqrt{2}}$ for representing the complex number $e^{\frac{\pi i}{4}} = \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}$. The real number $\frac{1}{\sqrt{2}}$ is approximated by the floating-point number $\mathtt{s|e|m} = \mathtt{0|1022|1865452045155277}$. Since $1 \leq \mathtt{e} \leq 1022$, $1 \leq \mathtt{m} \leq 2^{52} - 2$, and $\left\lfloor \frac{2^{105}}{2^{52}+\mathtt{m}} \right\rfloor \left(2^{52} + \mathtt{m}\right) = 6369051672525772 \left(2^{52} + \mathtt{m}\right) \geq 2^{105} - 2^{51}$, we know that if the other operand is the floating-point number `0|1|6369051672525772`, the result is incorrectly zeroized. One can pass the pair $\left(1022 \cdot 2^{52} + 1865452045155277, 2^{52} + 6369051672525772\right)$ as arguments of the emulated floating-point multiplication in Falcon and compare the result with the native floating-point multiplication to see the difference.

A.1.4 Is it Relevant to Falcon?

In previous section, we demonstrate that the emulated floating-point multiplication does not satisfy its claim where some non-zero floating-point numbers are zeroized. An immediate question is its impact to Falcon implementations. Among the functions in Falcon, we are interested in the complex FFTs in the signature generation where the inputs are polynomials with integer coefficients in $\left[-2^{15}, 2^{15}\right)$. After going through the tests for all the floating-point constants in the complex FFT, we find that 692 out of 2048 floating-point constants admit floating-point operands leading to incorrect zeroizations. Nevertheless, we model the floating-point addition, subtraction, and multiplication in CryptoLine, and show that all non-zero intermediate floating-point numbers have absolute values lie in

$$\left[2^{-476}, 2^{27}(2^{52} + 605182448294568)\right],$$

far away from triggering incorrect zeroizations.

A.1.4.1 Modeling with CryptoLine Instructions

We first model our own strings of CryptoLine instructions and start annotating CryptoLine programs with assertions and assumptions to transfer predicates between backend tools. The main difficulties are as follows:

- When to declare statements that should be proved by the backend proof systems?
- Which statements should be transferred between proof systems at a given point?

We do not know of any systematic approaches resolving the two difficulties. Nevertheless, we find the following constructions of intermediate symbols and annotations sufficient for verifying the range:

1. Construct the 128-bit product r of mantissas with the long multiplication.
2. Split the input into radix-25 representation with bit-field arithmetic, verify the correctness of the splitting with the SMT solver, and add the corresponding algebraic identities to the computer algebra system.
3. Compute the multi-limb product, verify its algebraic correctness with r in the computer algebra system, and add the corresponding boolean identities to the SMT solver.
4. Verify the remaining operations (zeroization, rounding, assembling) entirely with the SMT solver.

If we remove Steps 2. and 3., the SMT solver does not return a result (it does not find an instance disproving the properties, but it does not finish verifying over all the possible inputs).

A.1.4.2 Range-Checking

We develop our own range arithmetic in C++ computing the pre- and post-conditions to be verified. Once the pre- and post-conditions are computed for all the possible floating-point additions/subtractions/multiplications, we verify the correctness with CryptoLine. Typically, range-checking of floating-point arithmetic focus on upper-bounding the floating-point errors[1]. However, we need to derive non-trivial lower bounds of floating-point numbers for proving the non-smallness of the absolute values of non-zero floating-point numbers.

For two non-negative floating-point numbers $a.l \leq a.u$, we represent the subset $\{0\} \cup [a.l, a.u] \cup [-a.u, -a.l]$ as a structure with lower bound $a.l$ and upper bound $a.u$. Since the definition is symmetric for the positive and negative sides, we only store the positive bounds, and update the positive bounds throughout the entire computation. The zero values are included implicitly and we do not store its existence (it always exists in all the ranges). The range arithmetic of floating-point multiplication is straightforward as shown in Algorithm A.5. For

---

[1] For example, Frama-C [CKK+12] only shows that the floating-point number is upper-bounded by a floating-point number and lower-bounded by 0, which is useless for proving the non-smallness of the absolute values of non-zero floating-point numbers.

the floating-point addition/subtraction with the ranges a and b, we distinguish between two cases:

1. Case $a \cap b = \{0\}$: The upper bound is computed as the sum of upper bounds, and the lower bound is defined as the minimum of the absolute values of the differences between an upper bound and a lower bound from different ranges. In other words, the lower bound is defined as $\min(|a.u - b.l|, |b.u - a.l|)$.

2. Case $a \cap b = t \neq \{0\}$: The upper bound is also computed as the sum of upper bounds, and the lower bound is defined as the floating-point value with mantissa 0 and exponent field 52 smaller than the exponent field of $t.l$, since the smallest value occurs when subtracting two values with the real-value difference $2^{\mathtt{e}-1075}$ where `e` is the smallest exponent field of the two and choosing `e` as the exponent field of $t.l$ results in a worse case analysis. Since we have to shift the leading bit of mantissa to the 52nd position, the exponent field is subtracted by 52 and the mantissa becomes 2^{52}. By the definition of floating-point numbers, the leading bit of mantissa is stored implicitly. This is why we set the mantissa to 0 in the floating-point number representation.

Algorithm A.6 is an illustration of the range arithmetic of floating-point addition/subtraction. After replacing all the floating-point arithmetic with the range arithmetic in the FFTs of Falcon, we transform all the input-output tuples into pre- and post-conditions for the corresponding CryptoLine models. We then run CryptoLine to verify the conditions. Our CryptoLine verification shows that

- All the range arithmetic are correct within our modeling of floating-point addition, subtraction, and multiplication.

- All non-zero intermediate floating-point numbers have absolute values lie in

$$\left[2^{-476}, 2^{27}(2^{52} + 605182448294568)\right]$$

  when the input coefficients of FFTs are integers in $\left[-2^{15}, 2^{15}\right)$.

Table A.1 summarizes the verification time of the range conditions of floating-point additions and multiplications in Falcon's size-1024 complex FFT.

Algorithm A.5 Range arithmetic of floating-point multiplication.

```
Inputs: $a = (a.l, a.u), b = (b.l, b.u)$.
Output: $c = (c.l, c.u)$.
$c.l = a.l \cdot b.l$
$c.u = a.u \cdot b.u$
return $c$
```

Algorithm A.6 Range arithmetic of floating-point addition/subtraction.

```
Inputs: $a = (a.l, a.u), b = (b.l, b.u)$.
Output: $c = (c.l, c.u)$.
$t = a \cap b$.
if $t = \{0\}$ then
    $(d_0, d_1) = (|a.u - b.l|, |b.u - a.l|)$
    $c.l = \min(d_0, d_1)$
    $c.u = a.u + b.u$
    return $c$
$c.u = a.u + b.u$
$\mathtt{s|e|m} = t.l$
$c.l = \mathtt{s|(e - 52)|0}$
return $c$
```

Table A.1: Verification time (in seconds) of range conditions for a size-1024 complex FFT with $\mathbb{C} \cong \mathbb{R}[z]/\langle z^2 + 1\rangle$ by [Por19] and input polynomials drawn from $\left[-2^{15}, 2^{15}\right) \cap \mathbb{Z}$. Floating-point subtractions are regarded as floating-point additions in our interval arithmetic. FP stands for "floating-point."

Operation	# Instance	Second (avr. / total)
FP addition	767	0.297 886 / 228.478 732
FP multiplication	511	2.589 009 / 1 322.983 371

A.1.5 Equivalence Proofs

In this section, we briefly describe our implementations of floating-point multiplication and their equivalence proofs.

A.1.5.1 Our Implementations and The Claimed Behavior

Since there is a discrepancy between the emulated floating-point multiplications in Falcon and the claimed behavior, we implement our own assembly implementation honoring the following rules:

- It rounds the values correctly by experiment.
- Its output range is always zeros or normal floating-point values by formal verification. If the real number product is too small in absolute value, it returns a zero. If the real number product is too large in absolute value, the largest normal value is returned when the result is positive (smallest normal value is returned in the negative case).

We start with the assembly implementations in Falcon, which is much more optimized compared to the C reference implementations, and implement the above rules. This ensures that the output range is always a zero or a normal floating-point value when the inputs are zeros or normal floating-point values.

Comparisons to [Por19]. In the emulated floating-point multiplications in Falcon by [Por19], since the program does not handle infinities, one has to verify the correctness within a certain input range avoiding infinity outputs. The former forbids us to argue the correctness of the full range of zeros and normal floating-point values.

In addition, we also implement an emulated floating-point addition/multiplication in Jasmin essentially following the more readable (but slower) C reference implementation. In the follow-up section, we explain how to verify the equivalences of emulated floating-point multiplication implementations.

A.1.5.2 Equivalence Proofs in CryptoLine

We start with our CryptoLine model used for range-checking and add more annotations. Essentially, the majority of the effort is still about verifying the multi-limb arithmetic and transferring its correctness to the SMT solver. In principle, whenever we issue a multiplication, we prove its correctness in the computer algebra system, and add the corresponding boolean identities to the SMT solver. We apply the idea to proving the equivalence of our CryptoLine models and our assembly implementations, and the equivalence of our CryptoLine models and our Jasmin implementations. See Table A.2 for an overview of verification time of the equivalences. Since equivalence is transitive, we have equivalences between our assembly-optimized implementations and our

Jasmin implementations where the former is more optimized and the latter is more readable.

Table A.2: Verification time of equivalence proofs between Armv7-M implementations and our CryptoLine models.

Programming language	Verification time (in seconds)
Floating-point addition	
Jasmin	53.946 560
Assembly	59.863 976
Floating-point multiplication	
Jasmin	57.108 668
Assembly	5.333 913

A.1.6 Discussions

A.1.6.1 How the Discrepancy was Found?

The core of this paper is about modeling floating-point addition, subtraction, and multiplication with the domain-specific language CryptoLine, and its application to proving the lower bounds and upper bounds of non-zero intermediate floating-point numbers and the equivalences between implementations via software emulation. The whole paper is written in a way with concise logical reasoning so readers can follow more easily. However, the true story of the discovery is more disorganized than the story told in the paper.

The true story is that, we first wrote a model in CryptoLine and proved its equivalence with the emulated floating-point multiplication by [Por19]. With a much more readable model at hand, we were confounded by its correctness since it was inconsistent with our understanding of floating-point arithmetic. Our careful examinations eventually led to the C program extracting witnesses with incorrect zeroization, in the sense that the results of the emulated floating-point multiplication were different from the native floating-point multiplication on our laptop and the emulated floating-point multiplication by the Arm's tool-chain for Cortex-M4. After contacting the author of [Por19], we knew that experimentally, there were no such floating-point numbers but there was no formal proof. We later fixed our model, simplified it for range-checking, and verified the absence of non-zero floating-point numbers with absolute values the smallest normal positive floating-point number throughout the complex FFTs in the signature generation of Falcon. We also formalized the model for the floating-point addition, and the models of floating-point addition and multiplication were finally used for verifying the equivalences of implementations. We hope the true story will give more insights on how to use the tools.

A.1.6.2 The Validity of This Paper After Recent Uses of Fixed-Point Arithmetic

Recently, a fixed-point implementation for the complex FFT in the key generation was proposed by [Por23]. An immediate question is the validity of our findings in the emulated floating-point arithmetic. We would like to stress that, the roles of the complex FFTs are quite different in key generation and signature generation.

Key generation. We review the uses of complex FFTs in the key generation of Falcon as follows. We first generate short integer polynomials f and g, and solve for integer polynomials F and G satisfying

$$fG - gF = q \bmod (x^n + 1)\,.$$

Since the coefficients of F and G could be too large for efficient computation for the follow-up computation, we need to reduce the bit-size of the pair (F, G) with respect to the pair (f, g). This can be achieved by the Babai's reduction: we compute $k = \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil$ and subtract (kf, kg) from (F, G) where $f^* := f_0 - \sum_{i=1}^{n-1} f_i x^{n-i}$ is the adjoint of $f = \sum_{i=0}^{n-1} f_i x^i$. Obviously, if $fG - gF = q \bmod (x^n + 1)$, then $f\,(G - kg) - g\,(F - kf) = fG - gF = q \bmod (x^n + 1)$ and $(F - kf, G - kg)$ is a valid solution for the NTRU equation. For the quotient $\frac{Ff^* + Gg^*}{ff^* + gg^*}$ in $\mathbb{Q}[x]/\langle x^n + 1\rangle$, we instead compute them with the aid of complex FFTs in $\mathbb{C}[x]/\langle x^n + 1\rangle$. In [Por23], the author implemented the complex FFTs with scaled 64-bit fixed-point arithmetic and reduced the pair (F, G) several times instead of reducing it once with high-precision complex FFTs.

Signature generation. In the signature generation, the roles of the complex FFTs are quite different. Essentially, the sampler in Falcon converts the sampling task over the NTRU lattice into several one-dimensional sampling task and the complex FFTs are involved in this conversion. If one wants to replace the floating-point FFTs with scaled fixed-point arithmetic, one has to thoroughly revise the range analysis of the scaling, potentially use a much higher precision, and revise the security analysis from the implementation perspective. We have not seen effort from the community deploying the scaled fixed-point arithmetic and analyzing the accompanied security impact.

A.1.6.3 Possible Future Extensions

We briefly outline several possible future extensions of this paper.

Verifying additional constant-time emulations of floating-point arithmetic. This paper demonstrates the formal verification of the software emulation of floating-point addition, subtraction, and multiplication with respect to our CryptoLine models. Our approach extends to several interesting floating-point arithmetic, including negation, halving, and fused multiply-add/sub. Our approach also applies to other rounding rules. As for the floating-point division, it will be interesting to explore the formal verification of the bit-by-bit division by [Por19].

Applications to `ffLDL`* and `ffSampling`. In this paper, we verify the range of the complex FFTs with input integer polynomials. An immediate question is the applicability of our verification approach to the operations `ffLDL`* and `ffSampling` in the signature generation. For `ffLDL`*, it is a straight-line program with floating-point divisions so we can only verify the computation once floating-point division is verified. For `ffSampling`, it is built upon the one-dimensional discrete Gaussian sampler with a rejection loop. Therefore, CryptoLine along cannot verify this operation. We believe CryptoLine should be used as a plug-in of formal verification tools handling the rejection loop.

A.1.6.4 Applications to Other Lattice-Based Schemes

Our formal verification approach applies to several digital signature schemes. For ModFalcon [CPS$^+$20], since it also relies on the fast Fourier sampling from [DP16], one needs to apply FFT in a similar fashion as in Falcon's signature generation. For Mitaka [EFG$^+$22], there are two samplers proposed by [EFG$^+$22]: the hybrid sampler built upon the Gram-Schmidt orthogonalization with the aid of complex FFTs and the integer-arithmetic-friendly sampler built upon the integral Gram decomposition by [DGPY20]. For the former, our verification approach applies since one needs to apply complex FFTs. For the latter, integral Gram decomposition reduces to writing a positive integer as a sum of four squared integers and the fastest know algorithms are the randomized ones [PT18]. It seems difficult to find an unconditional deterministic algorithm for the problem [PT18, Section 5]. Therefore, it is unclear to us whether the integral version of Mitaka can be implemented securely and efficiently.

A.2 Paper: Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, Saber, and NTRU

Post-quantum cryptography requires a different set of arithmetic routines from traditional asymmetric cryptography such as elliptic curves. In particular, in each of the lattice-based NISTPQC Key Establishment finalists, every state-of-the-art optimized implementations for lattice-based schemes still in the NIST-PQC round 3 currently use different complex multiplications based on the Number Theoretic Transforms. We verify the NTT-based multiplications used in NTRU, Kyber, and Saber for both the AVX2 implementation for Intel CPUs and for the `pqm4` implementation for the ARM Cortex M4 using the tool CryptoLine. We extended CryptoLine and as a result are able to verify that in six instances multiplications are correct including range properties.

We demonstrate the feasibility for a programmer to verify his or her high-speed assembly code for PQC, as well as to verify someone else's high-speed PQC software in assembly code, with some cooperation from the programmer.

A.2.1 Introduction

Shor's algorithm [Sho97b] on a large-scale ("cryptographically relevant") quantum computer will solve today-intractable integer factorizations and discrete logarithms, hence breaking RSA and Elliptic Curve Cryptography (ECC) which make up almost all currently deployed asymmetric cryptography. The U.S. National Institute of Standards and Technology (NIST) has preemptively initiated a process (NISTPQC) to select new cryptosystems that withstand quantum computing. This research area is known as Post Quantum Cryptography (PQC). This process naturally divides into categories of digital signatures and key encapsulation mechanisms (KEMs) [NIS] and is currently in the third round with 7 finalists and 8 alternate candidates still competing [AASA^{+}20]. Schemes will be standardized when NISTPQC concludes. These will no doubt be important future computational workloads.

Since individual cryptographic operations are often slow themselves, and cryptography is then applied to much, much data, cryptography is always under a lot of pressure to be efficient. A common narrative has cryptographers developing new, faster, cryptographic primitives in reaction to this pressure. However, this is not an accurate depiction. Much of the actual speed comes from optimization research that actually takes a mathematical function then finds faster ways to compute that function. This research then feeds back into cryptographic designs. Performance pressure also results in a vastly more complex cryptographic software ecosystem. Herein we find many different intricate and often cutting-edge speedups using new mathematical algorithms or new micro-architecture-specific optimizations.

Every round-3 submission in NISTPQC includes hand-optimized software. Contrary to common impression, this is usually solidly faster than generic code compiled with a state-of-the-art "optimizing" compiler. Because PQC needs to survive quantum attacks, they also tend to be also more complex than pre-quantum public-key cryptography. Thus, post-quantum public-key software is usually even more complicated than pre-quantum public-key software like ECC, which can be complicated already. This aggravates any implementation problems. Because we shall be forced to roll PQC software out in a few years, we are also forced to ask ourselves: How do we minimize bug in PQC software? Traditional tests will miss many bugs, as exemplified by the following quote:

> Produced signatures were valid but leaked information on the private key. . . . The fact that these bugs existed in the first place shows that the traditional development methodology (i.e. 'being super careful') has failed.
>
> — "OFFICIAL COMMENT" `https://tinyurl.com/y5w46bde`

Testing only checks that an implementation is correct on a fixed set of selected inputs. There is no guarantee on untested inputs. Given the essentially infinite possibilities in inputs to PQC software, the proportion of tested inputs is always negligible. The obvious answer when we look for a better mousetrap is *formal verification*. This is a process wherein a conclusion can be reached that the software computes the correct outputs for *all* possible inputs—there are no rare (corner) cases that are handled incorrectly.

CryptoLine was developed to help programmers write correct cryptographic assembly programs. In particular, it is designed to verify arithmetic subroutines that make up the operations between elements of finite algebraic structures (rings, finite fields, elliptic curve groups). Such arithmetic subroutine is a common feature to most public-key cryptosystems. It is usually the case that if you run some tests in symmetric crypto, it will catch the bugs. In general, arithmetic operations that make up public-key crypto are harder to test, because a carry or an overflow — the kind of errors in arithmetic that happens when the programmer overlooks something — might happen very, very rarely, and we do not know if a potential attacker has a way to trigger such an event. Biham *et al* discuss scenarios where *hardware* bugs result in attacks [BCS16]. Software bugs can have the same effect, and the attackers can identify bugs in the programs using CryptoLine, just like us.

A.2.1.1 Our Contributions

First verification of NTT multiplications in assembly. We produce the first (semi-automatic) verification result for post-quantum crypto software. More precisely, we verify the highly complex polynomial multiplications based on the Number Theoretic Transform (NTT) in one instance of each in the NISTPQC Round 3 finalists Kyber, Saber, and NTRU. Our technique is applicable to any other software implementing lattice-based cryptosystems, such as NTRU Prime, LAC, or NewHope [LLZ+18, PAA+20, BBC+20], that also use NTT-based multiplications.

As illustrative examples, we picked the fastest software for one instance (parameter set) of each of the three NISTPQC lattice KEM finalists (to the best of our knowledge):

NTRU Intel AVX2: `ntt-polymul`[2] build 3e42ffa; ARM Cortex-M4: `pqm4`[3], build d26fee0, pull request `https://github.com/mupq/pqm4/pull/219`.

Kyber Intel AVX2: `PQClean`[4] build 688ff2f; ARM Cortex-M4: `pqm4`[3], build 944b3c3.

Saber Intel AVX2: `ntt-polymul`[2] build 3e42ffa; ARM Cortex-M4: Strategy A by [ACC+21][5].

As shown in Section A.2.7, the time used for these verification efforts is quite tolerable and would have been even less had the programmer been verifying his or her own code.

Extension of the CRYPTOLINE tool. We extend CRYPTOLINE, in particular we introduce *non-local compositional reasoning* in order to be able to finish all six instances. Without these extensions, the verification becomes either much slower or impossible.

A.2.1.2 Related Work

Faulty multiplication has been exploited as bug attacks [BCS16]. Formal verification on cryptographic programs aims to prove the absence of bugs and hence prevent such attacks. There exist many projects that (e.g. HACL [ZBPB17],

---

[2] `https://github.com/ntt-polymul/ntt-polymul`

[3] `https://github.com/mupq/pqm4`

[4] `https://github.com/PQClean/PQClean`

[5] `https://github.com/multi-moduli-ntt-saber/multi-moduli-ntt-saber`

Jasmin [ABB+17] and Fiat [EPG+19]) apply a correct-by-construction approach to build correct cryptography programs. This work is about verifying existing programs that have been not written in such a manner.

Various cryptography primitives have been formalized and manually verified in proof assistants (e.g. [Aff13, ANY12, AM07, MG07, MC13, App15, BPYA15, YGS+17]). We are trying to verify software in an automated or at least semi-automated fashion. Note that these are code "in the wild": the programs are written with an objective of speed or small size, and not with verifiability in mind.

The only verification result to our knowledge that is specifically conducted for post-quantum crypto today is EasyCrypt [BBF+21] which verifies protocols, not programs.

Many if not most of the verifications mentioned above use CoQ. We instead use the tool CryptoLine [TWY17, PTWY18, LST+19, FLS+19], described in detail in Section A.2.3.

Because Intel CPUs ("Haswell") and the ARM Cortex-M4 architectures were specified by NISTPQC as standard benchmarking platforms, there is so much literature on optimizing lattice-based schemes and also so much software available for these chips that we do not claim that the implementations we verified are the best or fastest. They were merely the fastest among the implementations conveniently at hand.

A.2.2 Preliminaries

We briefly describe the targets of our verification, then some mathematics involved (modular reductions, and the Number Theoretic Transform).

A.2.2.1 Kyber

The NISTPQC finalist candidate Kyber [ABD+20b] is a KEM based on the Module Learning With Errors (M-LWE) problem, using a dimension $\ell \times \ell$ module over the ring $R_q = \mathbb{F}_q[X]/\langle X^n + 1\rangle$, with $q = 3329$ and $n = 256$. Kyber is derived from a CPA-secure Public-Key Encryption (PKE) scheme via a Hofheinz–Hövelmanns–Kiltz CCA-transform [HHK17]. For a detailed PKE description, see [ABD+20b].

There is one $(\ell \times \ell) \times (\ell \times 1)$ matrix-to-vector polynomial multiplication (`MatrixVectorMul`) and zero, one, and two $(\ell \times 1)$ inner products of polynomials (`InnerProd`) in each of key generation, encapsulation, and decapsulation respectively. This is because decapsulation needs a full re-encryption. [ABD+20b] specifies that *we do all multiplications via incomplete NTT, and NTT results are in* *bit-reversed order.* All polynomial multiplications involve one random polynomial mod$q$ and one polynomial (s or s') with coefficients between $\pm\eta/2$, with some multiplicands already in NTT form. E.g., the public matrix A is sampled in (incomplete) NTT domain from a seed via the extendable-output function (XOF) `SHAKE128`.

Parameters. See Table A.3 for parameters: Module dimension ℓ and width of the centered binomial distribution η (twice the bound of the coefficients in "small" polynomials; rounding parameters (d_1, d_2) need not concern us), vary according to the parameter sets `kyber512`, `kyber768` (what we verified), and `kyber1024` (targeting NIST security levels 1, 3, and 5).

Table A.3: Kyber parameter sets.

name	l	(d_1, d_2)	$\eta(s\|s')$	$\eta(e\|e'\|e")$
`kyber512`	2	$(10, 4)$	6	4
`kyber768`	3	$(10, 4)$	4	4
`kyber1024`	4	$(11, 5)$	4	4

Table A.4: Saber parameter sets.

name	l	$T = 2^{\epsilon_T}$	η
`lightsaber`	2	2^3	10
`saber`	3	2^4	8
`firesaber`	4	2^6	6

A.2.2.2 Saber

The NISTPQC finalist candidate Saber [DKRV20] is a KEM based on the Module Learning With Rounding (M-LWR) problem, using a module of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$, with $q = 2^{13}$ and $n = 256$.

Saber KEM is also built on top of a CPA-secure PKE via the CCA-transform of Hofheinz-Hövelmanns-Kiltz [HHK17]. For an algorithmic description see [DKRV20].

There is one `MatrixVectorMul` in key generation; one `MatrixVectorMul` + one `InnerProd` in encapsulation; and one `MatrixVectorMul` + two `InnerProds` in decapsulation. All polynomial multiplications involve one random polynomial mod$q$ and one polynomial (marked s or s') with coefficients between $\pm\eta/2$.

In the Saber base ring $\mathbb{Z}_{2^{13}}$, 2 is not invertible and there are no appropriate principal roots, making it NTT-unfriendly. Accordingly, the specification samples the public matrix A in the polynomial domain.

Parameters. Module dimensions l and secret distribution parameters η (twice the bound of the coefficients in "small" polynomials; the rounding parameter T need not concern us) vary according to the parameter sets `lightsaber`, `saber`, and `firesaber` (targeting the NIST security levels 1, 3, and 5, cf. Table A.4). We verified `saber`.

A.2.2.3 NTRU

The NISTPQC finalist NTRU [CDH+20] is a KEM based on the hardness of the NTRU problems. It is based on NTRU as proposed by Hoffstein, Pipher, and Silverman in 1998 [HPS98]. It operates in the three polynomial rings $\mathbb{Z}_3[X]/\mathbf{\Phi_n}$, $\mathbb{Z}_q[X]/\mathbf{\Phi_n}$, and $\mathbb{Z}_q[X]/(\mathbf{\Phi_1}\cdot\mathbf{\Phi_n})$ with $\mathbf{\Phi_1} = (X-1)$ and $\mathbf{\Phi_n} = (X^{n-1} + X^{n-2} + \cdots + 1)$.

For algorithmic descriptions see [CDH+20]. NTRU achieves its CCA-secure KEM with a variation [SXY18] on the FO transform [FO99], avoiding having to re-encrypt the message during the decapsulation. NTRU is also not NTT-friendly by design, and one of the multiplicands in each product always has coefficients in $\{-1, 0, +1\}$.

Parameters. NTRU proposes 4 parameter sets (Table A.5) of which we verified `ntruhps2048509`.

Table A.5: NTRU parameter sets.

name	q	n
`ntruhps2048509`	$2048 = 2^{11}$	509
`ntruhps2048677`	$2048 = 2^{11}$	677
`ntruhrss701`	$8192 = 2^{13}$	701
`ntruhps4096821`	$4096 = 2^{12}$	821

A.2.2.4 Modular Reductions

Reductions modulo a small prime q is usually conducted through *signed Montgomery Reduction* [Sei18]: We pick a power of 2 as the "radix" $R > q$, and pre-compute $Q = 1/q \bmod R$. We can then compute $L = (A \bmod R)Q \bmod R$, then $(A - Lq)/R \equiv A/R \pmod{q}$. Since $R|(A - Lq)$, computing $(A - Lq)/R$ does not require a real division, and in fact only needs a high-limb multiplication (if available) when R has exactly the limb size.

In NTTs we are usually multiplying by known constants $\omega \pmod{q}$, and Seiler went further, introducing *Montgomery Multiplication* [Sei18]: Pre-compute $\omega' = \omega Q \bmod R$, then $b\omega$ can be computed as follows: $H = \lfloor b\omega/R \rfloor$ (*multiply, high*), then $L = b\omega' \bmod R$ (multiply, low), then $b\omega/R \equiv H - \lfloor Lq/R \rfloor \pmod{q}$ (again *multiply, high* and subtract).

Notice that the result of Montgomery reduction and multiplication $\pmod{q}$ is between $\pm q$, not $\pm q/2$. This is an example of *lazy reductions*. In high-speed implementations, the programmer never does any full reductions unless and until absolutely forced to.

A.2.2.5 The Number Theoretic Transform (NTT) and Butterflies

NTTs are critically important for speed in long multiplications. Classic works on integer multiplications [SS71, Für09, HvdH21] use them as basic blocks. NISTPQC 3rd round candidates Dilithium [ABD+20a], Falcon [PFH+20], and Kyber [ABD+20b] wrote NTTs into their specs to squeeze out extra efficiency improvements. NTRU [CDH+20], Saber [DKRV20], and NTRU Prime [BBC+20] can also use NTTs for speed [ACC+20, CHK+21].

Standard fast Fourier transform (FFT) and NTT. The “usual” radix-2 NTT/FFT means recursively using this ring isomorphism [CT65]:

$$\begin{aligned}\mathbb{F}[X]/\langle X^{2n} - c^2\rangle &\cong \mathbb{F}[X]/\langle X^n - c\rangle \times \mathbb{F}[X]/\langle X^n + c\rangle, \\ \sum_{i=0}^{2n-1} f_i X^i &\leftrightarrow \left(\sum_{i=0}^{n-1} (f_i + c f_{n+i}) X^i, \sum_{i=0}^{n-1} (f_i - c f_{n+i}) X^i \right),\end{aligned}$$

which holds if $2c$ is invertible. Considered as an “in-place” operation, starting with a size-$2n$ array of elements of $\mathbb{F}$ representing an element of $\mathbb{F}[X]/\left(X^{2n} - c^2\right)$ and ending with the bottom and top half of that array representing the elements of $\mathbb{F}[X]/(X^n - c)$ and of $\mathbb{F}[X]/(X^n + c)$ respectively, then with a little change of notation we may depict the map in Figure A.1 and its inverse map, up to a factor of 2, in Figure A.2. We refer to c as a *twiddle factor* (of the butterfly). If $2|n$ and $\sqrt{c} \in \mathbb{F}$ can be found we can repeat the process.

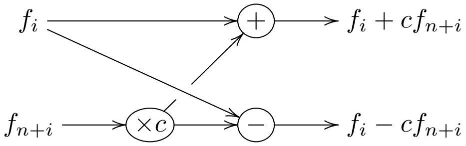


Figure A.1: Cooley–Tukey (CT) Butterfly.

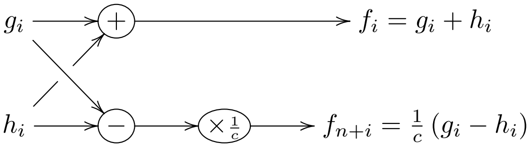


Figure A.2: Gentleman–Sande (GS) Butterfly.

As described by Cooley–Tukey, this only stops at linear factors, when we have reduced a polynomial multiplication in $\mathbb{F}[X]/\langle X^{2^k} - c\rangle$ to many independent multiplications in $\mathbb{F}$.

An FFT/NTT as described outputs in a "bit-reversed" order. When the NTT (FFT) is strictly to multiply two polynomials, we can ignore the different output order as long as the inverse NTT takes this into account.

"Twisted" FFT and NTT. Gentleman–Sande proposed a slightly different procedure [GS66] in which with the help of a such that $a^n = -1$ we apply recursively the following transformation

$$\frac{\mathbb{F}[X]}{\langle X^{2n} - 1\rangle} \cong \frac{\mathbb{F}[X]}{\langle X^n - 1\rangle} \times \frac{\mathbb{F}[X]}{\langle X^n + 1\rangle} \overset{X=aY}{\cong} \frac{\mathbb{F}[X]}{\langle X^n - 1\rangle} \times \frac{\mathbb{F}[Y]}{\langle Y^n - 1\rangle}.$$

Mapping $X = aY$ from $\mathbb{F}[X]/\langle X^n - c\rangle$ to $\mathbb{F}[Y]/\langle Y^n - 1\rangle$ is called *twisting*. Twisted (Gentleman–Sande) NTTs (FFTs) apply GS butterflies and its inverse apply CT butterflies.

One can see from Figures A.1 and A.2 that if Montgomery multiplication is used, starting from values bounded by $\pm q/2$, after ℓ layers of CT butterflies, the new values are bounded by $\pm\left(\ell + \frac{1}{2}\right) q$ whereas GS butterflies return values between $\pm 2^{\ell-1} q$. In general CT butterflies are better for lazy reduction, and as a result *some implementations do normal NTTs going forward and twisted NTTs in inverse so as to be able to use CT butterflies both ways.*

Principal roots and incomplete NTTs. To split $\mathbb{F}[X]/\langle X^n - c\rangle$ and repeat it k times requires that there is an $a \in \mathbb{F}$ such that $a^n = c$. Obviously, we need $2^k|n$, and when $c = 1$, we need a to be a *principal root* of 1: Let $[n]_q = \sum_{i=0}^{n-1} q^i$ be the q-analog of n. A *principal* nth root of unity ω is an nth root of unity satisfying the orthogonality $[n]_{\omega^i} = 0$ for $1 \leq i < n$ [Für09, HvdH21]. The existence of a principal root $\pmod m$ means that $n|(p-1)$ for all primes $p|m$. This definition coincides with *primitive* roots when m is prime.

If we stop short in our sequence of mappings prior to reaching linear factors, we have what is called an “incomplete” NTT/FFT and we are left with modular multiplications of low-degree polynomials. Sometimes we are forced to stop because the appropriate roots do not exist, sometimes because of efficiency considerations.

A.2.3 The CryptoLine tool

CryptoLine [TWY17, PTWY18, LST+19, FLS+19] is a tool intended for a programmer to verify his (or her) own arithmetic programs. It was developed with the idea that a programmer need not write within a fixed framework or depend on the whims of the compiler, as in [EPG+19]. Instead, the programmer codes any which way as desired. The main program of CryptoLine is written in OCaml while some subsidiary scripts are in Python.

A.2.3.1 The CryptoLine Language

CryptoLine is a domain-specific language for modeling cryptographic assembly programs and their specifications [PTWY18]. CryptoLine is a strongly-typed language. Constants in CryptoLine are associated with a type. Variables must have specific types according to *declarations*. Operations can only be between specific types. All casts are explicit. See Figure A.3 for a summary the construction of atoms and variables.

$$\begin{array}{rcl} \mathit{Num} & := & \cdots \mid \mathsf{-2} \mid \mathsf{-1} \mid \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \cdots \\ \mathit{Const} & := & \mathit{Num}@\mathit{Type} \\ \mathit{Var} & := & \cdots \mid x \mid y \mid z \mid \cdots \\ \mathit{Atom} & := & \mathit{Var} \mid \mathit{Const} \end{array}$$

Figure A.3: CryptoLine atoms and variables.

Let w be a positive integer. The type $\mathsf{uint}\ w$ comprises unsigned integers denoted by bit strings of length w. Similarly, $\mathsf{sint}\ w$ are signed integers denoted by bit strings of length w (2's complement). So uint w denotes integers in $[0, 2^w)$ and sint w denotes integers in $\left[-2^{w-1}, 2^{w-1}\right)$. bit is short for uint 1. So in these verifications, we deal with (mostly) uint32, sint32, uint16, sint16, and bit.

General instructions. Figure A.4 shows the syntax of CryptoLine. An arithmetic instruction retrieves values from *sources* and stores results in *destinations*. For example, mov v a copies the source a to the destination v; while depending on the value of c, cmov v c a_0 a_1 stores either the value of sources a_0 or a_1 in the destination v.

To model arithmetic in cryptographic assembly programs, many instructions involve flags. For example, adds c d a b means to take two atomic inputs a, b of (equal) size w, add them into an integer of size $w + 1$, then splits the top bit off into the first destination variable (c, the carry) and the second destination variable (d, the destination register of the instruction). "Short" instructions not covering the apparent output range requires that overflows do not happen. For example, add d a b says to add atomic inputs a, b of the same size w and checks that the result fits in the destination variable d of the same size w. Recall that signed and unsigned integers have different bounds when they are of size w. CryptoLine type system infers the types of sources to decide if their sum is representable by the destination variable.

In addition to additions and subtractions, to deal with multi-word arithmetic, CryptoLine also includes multi-word constructs for example, those that split (spl) or join (join) words, as well as multi-word shifts (cshl). Finally, there are long multiplications (mull) as well as their "short" version (mul). When a signed integer of size $w+v$ is split into two integers of size w and v respectively, the more significant destination is signed but the less significant destination is unsigned. CryptoLine type system again infers types of destination variables to ensure all arithmetic computation is within proper bounds.

Finally, the cast instruction casts the source to a designated type. CryptoLine checks whether integers in all executions are within the bounds of the designated types. Type inference and bound checking are useful in detecting overflows and underflows. They are especially helpful when a signed Montgomery reduction is used. Particularly, both signed and unsigned integers coexist after a signed Montgomery reduction in various NTT implementations. There must not be overflows nor underflows for all the arithmetic instructions in NTTs.

$$
\begin{array}{rcl}
\mathit{Inst} & := & \mathsf{mov}\ \mathit{Var}\ \mathit{Atom} \\
& | & \mathsf{cmov}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{add}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{adds}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{adc}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{adcs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{sub}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{subs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{sbb}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{sbbs}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{mul}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{mull}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{shl}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \\
& | & \mathsf{spl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Num} \\
& | & \mathsf{cshl}\ \mathit{Var}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom}\ \mathit{Num} \\
& | & \mathsf{join}\ \mathit{Var}\ \mathit{Atom}\ \mathit{Atom} \\
& | & \mathsf{cast}\ \mathit{Var@Type}\ \mathit{Atom} \\
& | & \mathsf{assert}\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
& | & \mathsf{assume}\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
& | & \mathsf{cut}\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
& | & \mathsf{ghost}\ \mathit{Var@Type}\ :\ \mathit{APred}\ \&\&\ \mathit{RPred} \\
\mathit{Decl} & := & \mathit{Type}\ \mathit{Var} \\
\mathit{Prog} & := & \mathit{Decl}^{*}\ \mathit{Inst}^{*}
\end{array}
$$

Figure A.4: CryptoLine syntax.

Asserts and assumes. As a modeling language, CryptoLine also provides special instructions for verification purposes. The assert P && Q instruction checks if both the algebraic predicate P and range predicate Q are true among all executions. An *algebraic predicate* is a conjunction of equations or modular equations. A *range predicate* is an arbitrary Boolean formula over comparisons, equations, and modular equations. In CryptoLine, algebraic predicates are verified by the CAS; and range predicates are verified by the SMT solver. See Figure A.5 for a summary of the definitions of expressions and predicates. When programmers would like to check if their programs compute as expected, they can add assert instructions with intended algebraic or range predicates at suitable locations. CryptoLine will verify these predicates automatically.

$$\begin{array}{rcl}
\mathit{Exp} & := & \mathit{Atom} \\
 & | & \mathit{Exp}\ +\ \mathit{Exp} \\
 & | & \mathit{Exp}\ -\ \mathit{Exp} \\
 & | & \mathit{Exp}\ \times\ \mathit{Exp} \\
\mathit{APred} & := & \mathit{APred} \wedge \mathit{APred} \\
 & | & \mathit{Exp} = \mathit{Exp} \\
 & | & \mathit{Exp} \equiv \mathit{Exp} \bmod [\mathit{Exp}, \mathit{Exp}, \ldots, \mathit{Exp}] \\
\mathit{RPred} & := & \mathit{Exp} < \mathit{Exp} \\
 & | & \mathit{Exp} = \mathit{Exp} \\
 & | & \mathit{Exp} \equiv \mathit{Exp} \bmod \mathit{Exp} \\
 & | & \mathit{RPred} \wedge \mathit{RPred} \\
 & | & \neg \mathit{RPred}
\end{array}$$

Figure A.5: CryptoLine expressions and predicates.

The `assume` P && Q instruction on the other hand imposes the algebraic predicate P and range predicate Q on all executions. Effectively, P and Q become premises after the `assume` instruction. `assume` P && Q are used to summarize previously verified predicates in `assert` P && Q to save verification time. Another frequent use of `assume` is to pass verified predicates between CAS and SMT solvers. Recall that different techniques are applied to verify algebraic and range predicates in CryptoLine respectively. When a range predicate is verified, the established property is unknown to CAS and vice versa. CAS nevertheless can be informed of verified range predicates with `assume`. Consider the following sequence of instructions

`assert true` && Q `assume` Q && `true`.

CryptoLine first verifies the predicate Q with the SMT solver in the `assert` instruction. If Q holds for all executions, the predicate is passed to the CAS via the `assume` instruction. Particularly, the *short* `add` $d\ a\ b$ instruction is implicitly `adds` $c\ d\ a\ b$ followed by `assert true` && $c = 0$ and `assume` $c = 0$ && `true`. CryptoLine first asserts the carry $c = 0$ with SMT solver and assumes $c = 0$ in CAS.

A.2.3.2 Compositional Reasoning with Ghost Variables and Cuts

Compared with programs for field or group operations in elliptic curve cryptography, NTT implementations are significantly larger. Montgomery ladderstep in Curve25519 takes four 255-bit field elements and one 256-bit exponent as its inputs. There are roughly 1.3×10^3 input bits. `kyber768` NTT, on the other hand, takes 256 12-bit coefficients ($\approx 3.0 \times 10^3$ bits) as the inputs. Saber NTT takes 256 13-bit coefficients ($\approx 3.3 \times 10^3$ bits). NTRU2048509 takes 509 11-bit coefficients ($\approx 5.6 \times 10^3$ bits). Since input bits of various NTTs are multiples of those in Montgomery ladderstep, the information to be processed is significantly larger. It is perhaps natural to expect much longer cryptographic programs in post-quantum cryptography.

Lengthy cryptographic programs pose new challenges to formal verification. Since verification aims to establish program correctness for all inputs, an extra input bit can double the number of inputs. Longer computations induced by lengthy programs also increases the number of program states for analysis. The infamous state explosion problem severely limits the applicability of formal verification in practice. To verify various NTT implementations formally, new techniques are added to improve the scalability of CRYPTOLINE significantly.

Ghost variables. Computation in a cryptographic program often runs in clearly demarcated stages. The verifier often needed to specify a mid-condition to summarize the computation "so far" by stages as well. Sometimes, one would like to specify the mid-condition by relating variable values before and after the stage. When the program computes "in place", variable values prior to the stage would be overwritten. Ghost variables in CRYPTOLINE allow verifiers to store values for later reference. Consider, for instance, the computation of NTT by levels. For efficiency, cryptographic assembly programs often load data at level 0 and compute in registers for later levels. At the beginning of each level, verifiers can store register values in ghost variables. At the end of the level, verifiers specify the relations between ghost variables and registers in the mid-condition. The computation can then be verified by levels.

Cuts. Compositional reasoning is a divide-and-conquer technique widely used for ameliorating the state explosion problem in formal verification. The basic idea is to reduce large verification problems into smaller problems. If small problems can be solved, large problems are verified as well. The question, of course, is how to perform such a reduction soundly to avoid incorrect verification results.

CRYPTOLINE provides a simple mechanism to reason about cryptographic programs compositionally. The `cut` P && Q instruction allows CRYPTOLINE to verify a program by parts. Let Π_0 and Π_1 be sequences of instructions. Consider the following CRYPTOLINE program

$$\Pi_0 \quad \mathsf{cut}\ P \mathbin{\&\&} Q \quad \Pi_1.$$

CryptoLine transforms the program into the following two programs

$$\Pi_0 \quad \mathsf{assert}\ P \mathbin{\&\&} Q \qquad\qquad \text{and} \qquad\qquad \mathsf{assume}\ P \mathbin{\&\&} Q \quad \Pi_1.$$

In other words, CryptoLine first verifies the predicates P and Q at the end of Π_0. If both predicates hold, CryptoLine then uses P and Q as premises to verify Π_1. The program $\Pi_0 \quad \Pi_1$ is divided into two smaller programs: $\Pi_0 \quad \mathsf{assert}\ P \mathbin{\&\&} Q$ and $\mathsf{assume}\ P \mathbin{\&\&} Q \quad \Pi_1$. If any of them fails to verify, the original program $\Pi_0 \quad \Pi_1$ fails as well. The reduction is clearly sound. Both predicates P and Q are verified before they are assumed as premises. Effectively, P and Q can be seen as a summary of the computation in Π_0. The sub-program Π_1 is in turn verified with respect to the summary. Observe that $\mathsf{cut}\ P \mathbin{\&\&} Q$ divides a program $\Pi_0 \quad \Pi_1$ into two sub-programs Π_0 and Π_1 by locality. Since computational dependency often coincides with code locality, the predicates P and Q suffice to summarize the computation of Π_0 and verify the computation of Π_1. We therefore call the condition $P \mathbin{\&\&} Q$ in the cut instruction as a *mid-condition*.

Despite of its applicability in verification, classical compositional reasoning with cuts is insufficient for verifying NTT implementations for post-quantum cryptosystems effectively. For lattice-based cryptosystems, input polynomials for NTTs have degrees in hundreds or even thousands. Take the 7-level NTT used in `kyber768` as an example. Since different levels have different patterns of computation, implementations naturally compute `kyber768` NTT by levels. A naïve decomposition for `kyber768` NTT implementations would be as follows.

$$\begin{array}{rl}
\Pi_0 & \text{(* first level *)} \\
\mathsf{cut}\ P_0 \mathbin{\&\&} Q_0 & \text{(* summary of first level *)} \\
\vdots & \\
\Pi_5 & \text{(* sixth level *)} \\
\mathsf{cut}\ P_5 \mathbin{\&\&} Q_5 & \text{(* summary of sixth level *)} \\
\Pi_6 & \text{(* seventh level *)}
\end{array}$$

Using cuts, `kyber768` NTT is divided into seven sub-programs by levels; each level has 256 12-bit coefficients. Verifying all 256 coefficients are computed correctly at each level is certainly better than verifying seven levels of computation. Yet it is far from ideal. In `kyber768` NTT, recall that a coefficient at level ℓ depends only on two coefficients at level $\ell - 1$ for $0 < \ell \leq 6$. If `kyber768` NTT implementations could be decomposed by dependencies, it would further reduce the size of verification problems and improve the efficiency of formal verification.

Such decomposition however are not attainable through classical compositional reasoning with `cut`s. Since `kyber768` NTT implementations compute by levels, a coefficient may be computed long after its dependent coefficients were computed. Code locality is therefore different from computation dependency. The `cut` instruction on the other hand requires the correspondence between code locality and computation dependency. Classical compositional reasoning with `cut`s cannot further decompose the `kyber768` NTT computation at each level. More sophisticated compositional reasoning is needed.

To verify NTT implementations in lattice-based post-quantum cryptosystems, we extend the CryptoLine `cut` instruction to support non-local compositional reasoning. To see how it works, consider a `kyber768` NTT implementation again as follows.

```
                  $\Pi_{0,0}$                       (* 1st pair of coefficients in first level *)
cut   0 : $P_{0,0}$ && $Q_{0,0}$                     (* summary of 1st pair *)
                  ⋮
                  $\Pi_{0,127}$                     (* 128th pair of coefficients in first level *)
cut 127 : $P_{0,127}$ && $Q_{0,127}$                 (* summary of 128th pair *)
                  $\Pi_{1,0}$                       (* 1st pair of coefficients in second level *)
cut 128 : $P_{1,0}$ && $Q_{1,0}$ prove with $0, 64$  (* summary of 1st pair *)
                  ⋮ $\Pi_{6,127}$                   (* seventh level *)
cut 895 : $P_{6,127}$ && $Q_{6,127}$                 (* summary of 128th pair *)
```

Now the NTT implementation is decomposed by coefficient pairs. Additionally, each `cut` instruction is assigned to a number for reference. When a `cut` instruction is verified, our extension allows verifiers to add more premises by cut numbers. In the above example, the first coefficient pair of the second level depends on the first and sixty-fifth coefficient pairs of the first level. We therefore add the corresponding cut numbers as additional premises to verify the coefficients in the second level of `kyber768` NTT. Other coefficient pairs are verified similarly. Our extension admits more refined compositional reasoning. It allows us to verify NTT implementations with several hundreds of input coefficients effectively.

mov b 0@bit
cut $0 : b = 0 \mathbin{\&\&} b = 0$
mov b 1@bit
cut $1 : b = 1 \mathbin{\&\&} b = 1$
cmov x b 3142@uint16 2718@uint16
cut $2 : x = 42 \mathbin{\&\&} x = 42$ prove with $0, 1$

Figure A.6: Before SSA transformation.

mov b_0 0@bit
cut $0 : b_0 = 0 \mathbin{\&\&} b_0 = 0$
mov b_1 1@bit
cut $1 : b_1 = 1 \mathbin{\&\&} b_1 = 1$
cmov x_0 b_1 3142@uint16 2718@uint16
cut $2 : x_0 = 42 \mathbin{\&\&} x_0 = 42$ prove with $0, 1$

Figure A.7: After SSA transformation.

In cryptographic assembly implementations, registers are necessarily reused in computation. Care must be taken to avoid unsound verification results. In Figure A.6, the bit variable b is set to 0 and the computation is summarized by cut 0. Then b is set to 1 and summarized by cut 1. The conditional assignment then sets the variable x to either 3142 or 2718 by the value of b. At cut 2, CryptoLine is asked to verify whether x is 42 with premises $b = 0$ (from cut 0) and $b = 1$ (from cut 1). Since the conjunctive premise $b = 0$ and $b = 1$ is always false, cut 2 is verified vacuously. That is, x is 42. This is unsound.

To avoid unsoundness, our extension transforms CryptoLine programs to the static single assignment (SSA) form before formal analysis (Figure A.7). The SSA transformation allows our analysis to identify different versions of the same variable uniquely. After SSA transformation, the premises for cut 2 are $b_0 = 0$ and $b_1 = 1$. Their conjunction is not false. CryptoLine fails to verify $x = 42$ and finds a counterexample easily. In fact, x_0 is always 3142, and independent of b_0 as expected.

Techniques of using CryptoLine. Using `itrace.py` and `to_zdsl.py`, a CryptoLine program can be obtained rather easily. Verifiers need to annotate the CryptoLine program with a proper pre-condition, post-condition, and possibly several mid-conditions. These conditions can be derived with the help of programmers or by inspecting the program. Verifiers may choose to specify these conditions with algebraic or range predicates. Since CryptoLine employs different techniques to verify different predicates, its effectiveness varies by the choice of verifiers' specification.

Range predicates are verified by the SMT solver. Very roughly, CryptoLine translates programs into Boolean circuits whose free inputs are the input parameters of `main`. The pre-condition is an additional constraint on free inputs. The *negation* of the post-condition is another constraint on the Boolean circuits. The verification tool then calls an SMT solver to check if the Boolean circuit with constraints is satisfiable. If the answer is "SAT", the negation of the post-condition holds for certain input values satisfying the pre-condition. The verification fails (and we can output those inputs as counterexamples). If the answer is "unSAT", the SMT solver has determined that there are no input value satisfying the pre-condition but falsifying the post-condition at the end of the program. The verification succeeds.

This is a well-established technique widely used in hardware and bit-accurate software verification. Verifying range predicates requires minimal human guidance, verifiers are recommended to write range predicates in general. SMT solver however does have limitations. For instance, it is widely known that SMT solvers are ineffective in verifying non-linear computation. Cryptographic programs almost surely perform non-linear computation. A more effective verification technique is needed for such programs.

CryptoLine employs a CAS to verify algebraic predicates. Particularly, non-linear equations and modular equations can be verified easily by the algebraic technique. The verification tool essentially translates every CryptoLine instruction into polynomial equations. For example, adds $c\ d\ a\ b$ is translated to $2^w c + d = a + b$ and $c(1-c) = 0$ when a and b are unsigned integers of size w; cmov $d\ c\ a\ b$ is translated to $d = ca + (1-c)b$. Note that all possible executions of the instructions adds $c\ d\ a\ b$ or cmov $d\ c\ a\ b$ are the roots of corresponding polynomial equations. A CryptoLine program is thus translated to a set of polynomial equations. All program executions are also roots of the set of polynomial equations. To verify if all program executions satisfy the post-condition, it suffices to verify if all the roots of polynomial equations for the program are also the roots of the polynomial equations in the post-condition. CryptoLine calls a CAS to solve this algebraic problem. Instead of logical techniques, non-linear computation is thus verified by algebraic techniques.

Verifiers are recommended to write algebraic predicates to verify non-linear computations. Algebraic predicates nevertheless are very restrictive. They do not allow comparison and must be conjunctive. It is sometimes necessary to combine both CAS and SMT solvers to verify conditions. Verifiers need to be creative to pass information between the two techniques via `assert` and `assume`. Human guidance is still needed during verification.

Consider, for example, the signed Montgomery reduction in Section A.2.2.4. We have $A - Lq = A - ((A \bmod R)Q \bmod R)q \equiv A - AQq \equiv 0 \pmod{R}$. Computing $A - Lq$ requires a full multiplication, and its value is stored in two registers r_H and r_L where the low-limb register r_L is always zero. Because of non-linear computation, SMT solver shows $r_L = 0$ but requires some effort. CAS easily shows $r_L \equiv 0 \pmod{R}$ but not $r_L = 0$ on the other hand. Since the radix R is precisely the word size, $r_L \equiv 0 \pmod{R}$ is actually $r_L = 0$. Verifiers can safely assume $r_L = 0$ after CAS asserts $r_L \equiv 0 \pmod{R}$.

A.2.3.3 Walkthrough: How the AVX2 `kyber768` NTT is Verified

Notations. NTT layers go up from 0, and inverse NTT (iNTT) layers count down to 0.

- $F = \sum_{k=0}^{n-1} f_k X^k \in \mathbb{Z}_q[X]$ is the polynomial we began with. If we central-reduce F first before the NTT, the result is marked with a "hat" ($\hat{F}$, $\hat{f}_k$).
- After NTT level i, the jth polynomial is $G_{i,j} = \sum_{k=0}^{n/2^{i+1}-1} g_{i,j,k} X^k$ for $0 \le j < 2^{i+1}$.
- $\zeta_{i,j}$ are the roots of unity used at the end of level i (counting up).
- $\mathbb{Z}_q[X]/\langle X^{n/2^L} - \zeta_0 \rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^{n/2^L} - \zeta_{2^L-1} \rangle$ contains the NTT result, so $\zeta_{L-1,j} = \zeta_j$, where $0, \ldots, (L-1)$ number the L levels.
- The NTT result comprises polynomials $P_j = \sum_{k=0} p_{j,k} X^k$ (we see the array of $p_{j,k}$'s).
- After iNTT level i, the jth polynomial is $H_{i,j} = \sum_{k=0}^{n/2^i-1} h_{i,j,k} X^k$ for $0 \le j < 2^i$.
- $\overline{F}$ is the result of the inverse NTT.

We will first give an overview of what is involved in verifying a high-speed NTT in assembly — handwritten by somebody else — with this walk-through.

The AVX2 `kyber768` NTT is chosen because it is simplest and illustrates our points well.

Starting from the executable, a running trace of a subroutine is extracted to be verified, using the script `itrace.py` that calls `gdb`. The extracted trace looks like the following:

```
$ itrace.py test ntt PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas
$ more  PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas

#PQCLEAN_KYBER768_AVX2_polyvec_ntt:
:
# [some bookkeeping information]
:
vmovdqa (%rsi),%ymm0            #! EA = L0x5555556395e0; Value = 0x0d010d010d010d01; \
                                                                  PC = 0x55555556eb4f
vpbroadcastq 0x140(%rsi),%ymm15 #! EA = L0x555555639720; Value = 0x7b0a7b0a7b0a7b0a; \
                                                                  PC = 0x55555556eb53
vmovdqa 0x100(%rdi),%ymm8       #! EA = L0x7fffffffb080; Value = 0xffff0000ffff0001; \
                                                                  PC = 0x55555556eb5c
:
vpbroadcastq 0x148(%rsi),%ymm2  #! EA = L0x555555639728; Value = 0xfd0afd0afd0afd0a; \
                                                                  PC = 0x55555556eb7c
vpmullw %ymm15,%ymm8,%ymm12     #! PC = 0x55555556eb85
:
vpmulhw %ymm2,%ymm8,%ymm8       #! PC = 0x55555556eb99
:
vmovdqa (%rdi),%ymm4            #! EA = L0x7fffffffaf80; Value = 0x0000ffff00000000; \
                                                                  PC = 0x55555556eba9
:
vpmulhw %ymm0,%ymm12,%ymm12     #! PC = 0x55555556ebbc
:
vpaddw %ymm8,%ymm4,%ymm3        #! PC = 0x55555556ebcc
vpsubw %ymm8,%ymm4,%ymm8        #! PC = 0x55555556ebd1
:
:
```

`test` is a test program compiled to use the routine in question. Most instructions start with `vp` indicating the AVX2 instruction set. We note that the above code loads two sets of 64 coefficients into `%ymm4-7` and `%ymm8-11`, then a set of twiddle factors (in Montgomery form) into `%ymm15` and starts computing butterflies using Montgomery multiplications. The program actually does 4 butterflies at a time, the snippet above only contains code pertaining to just one butterfly (the dots here as below stand for cut material).

We put a set of translation rules on top of a running trace and then run another script, `to_zdsl.py`. The rules to the above program looks like

```
#! $1c(%rsi) = %%EA
#! (%rsi) = %%EA
#! $1c(%rdi) = %%EA
#! (%rdi) = %%EA
#! %ymm$1c = %%ymm$1c
#! vpbroadcastq $1ea, $2v -> mov $2v_0 $1ea;\nmov $2v_1 $1ea[+2];\nmov $2v_2 $1ea[+4]; ...
```

```
#! vmovdqa $1ea, $2v -> mov $2v_0 $1ea;\nmov $2v_1 $1ea[+2];\nmov $2v_2 $1ea[+4]; ...
#! vmovdqa $1v, $2ea -> mov $2ea $1v_0;\nmov $2ea[+2] $1v_1;\nmov $2ea[+4] $1v_2; ...
```

The initial lines specify variables. `#! $1c(rdi)=%%EA` and `#! (rdi)=%%EA` map register-indirect-offset-addressed memory to more memory variables. Each lines after that describes an instruction. For example, `vpbroadcastq $1ea, $2v` means to broadcast the 64-bit memory location `$1ea` into each 64-bit limb of the target register `$2v`; `vpmullw` means to multiply each pair of matching 16-bit limbs in the two source registers into the corresponding limbs in the target register, etc. Note we have to read the source and understand what is going on to annotate the program appropriately. For example the multiplication instructions require special care:

```
#! vpmullw $1v, $2v, $3v -> smull mulH$2v_0 mulL_0 $1v_0 $2v_0;\n ...
#! vpmulhw %%ymm0, $1v, $2v -> smull mulH_0 red_0 $1v_0 ymm0_0;\n \
  \nassert true && red_0 = mulLymm_0;\nassume red_0 = mulLymm_0 && true;\n ...
#! vpmulhw $1v, $2v, $3v -> smull mulH_0 mulL$2v_0 $2v_0 $1v_0;\n ...
```

In CryptoLine, multi-limb multiplications always return unsigned lower parts, but we are using signed integers throughout, so in the translation rules for `vpmullw`, we need to typecast `@sint16` for each limb. A high-limb multiplication is often troublesome either with the matching lower-limb multiplication somewhere else in the code, or something assumed about the lower limb. Here, in a signed Montgomery multiplication, what is assumed is that particular pairs of unused lower-limbs are equal, and we can translate appropriately as `%ymm0` has always 16 qs. One can find this in the code, captured in the CryptoLine program by `vmovdqa (%rsi),%ymm0 #! EA = L0x5555556395e0;` and (see below) `mov L0x5555556395e0 ( 3329)@sint16; ...`, allowing us to annotate correctly.[6]

At this point, the script `to_zdsl.py` converts each actual CPU instruction into one or more lines in CryptoLine instructions, usually the latter in AVX2 code.

```
$ to_zdsl PQCLEAN_KYBER768_AVX2_polyvec_ntt.gas > PQCLEAN_KYBER768_AVX2_polyvec_ntt.cl

(* #! -> SP = 0x7fffffffa358 *)
:
(* many bookkeeping instructions deleted *)
:
(* vmovdqa (%rsi),%ymm0                   #! EA = L0x5555556395e0; ...
mov ymm0_0 L0x5555556395e0;
mov ymm0_1 L0x5555556395e2;
:
mov ymm0_f L0x5555556395fe;
```

---

[6] This is a benefit of handwritten assembly; equivalent intrinsics compiled code would migrate the q values from register to register over the course of the whole program, making our annotations much harder.

```
(* vpbroadcastq 0x140(%rsi),%ymm15       #! EA = L0x555555639720; ...
mov ymm15_0 L0x555555639720;
mov ymm15_1 L0x555555639722;
:
.............
```

After some irrelevant bookkeeping instructions, each vector instruction splits into 16× word-sized (16-bit) actions by our translation rules. Now, we set down what the constants (copied from source code) are, the entering conditions (inputs, assumptions/pre-conditions), and the concluding conditions (outputs, requirements/post-conditions). Again this requires understanding what the code does, and some scripts to generate the annotations.

```
proc main (
sint16 f000, sint16 f001, sint16 f002, sint16 f003,
...
sint16 f252, sint16 f253, sint16 f254, sint16 f255
) =
{
true && and [
(-3329)@16 <s f000, f000 <s (3329)@16, (-3329)@16 <s f001, f001 <s (3329)@16,
...
(-3329)@16 <s f254, f254 <s (3329)@16, (-3329)@16 <s f255, f255 <s (3329)@16
]
}
(*********** initialization ***********)
mov L0x7fffffffaf80 f000; mov L0x7fffffffaf82 f001;  mov L0x7fffffffaf84 f002;
...
mov L0x7fffffffb17e f255;
```

We declare the entering conditions. Each "condition" actually comprises two specifications: an algebraic part, to be checked with a Computer Algebra System (CAS; defaults to Singular but can be Magma, Mathematica, or Maple), and a range part, to be checked using an SMT (Satisfiability Module Theory) solver, here via BOOLECTOR. In the preamble above, the first `true` specifies that there are no restrictions algebraically on the input array, but the second portion restricts each entering polynomial coefficients to be between $\pm q$. Then "initialization" assigns each starting 16-bit limbs in memory, represented by `L(hex address)`, to an input variable `f###`.

```
(*********** constants ***********)
mov L0x5555556395e0 (  3329)@sint16; mov L0x5555556395e2 (  3329)@sint16;
...
mov L0x555555639600 ( -3327)@sint16; mov L0x555555639602 ( -3327)@sint16;
...
mov L0x555555639620 ( 20159)@sint16; mov L0x555555639622 ( 20159)@sint16;
...
mov L0x555555639adc (    32)@sint16; mov L0x555555639ade (    32)@sint16;
(*********** ghost polynomial ***********)
ghost x@bit, inp_poly@bit : inp_poly**2 =
f000*(x**0) + f001*(x**1) + f002*(x**2) + f003*(x**3) +
```

```
...
f252*(x**252) + f253*(x**253) + f254*(x**254) + f255*(x**255)
 && true;

(* main body of program goes here ... *)
```

Each PQClean NTT uses an array of twiddle factors that already resides in memory, and we copy the numbers (as in the snippet) directly from the source, inserted using a `python` script. The `ghost` polynomial is a compositional reasoning gadget that combines the entering coefficients into one entity (cf. Section A.2.3.2). After level 0 is completed, we fill in the conditions according to the description of the NTT in Section A.2.2.5.

```
:
(*********** SUMMARY OF LEVEL 0 ***********)

cut and [
eqmod (inp_poly**2)
(L0x7fffffffaf80*(x**0) + L0x7fffffffaf82*(x**1) + L0x7fffffffaf84*(x**2) +
:
 L0x7fffffffb07c*(x**126) + L0x7fffffffb07e*(x**127))
[3329, x**128 - (1729)],
eqmod (inp_poly**2)
(L0x7fffffffb080*(x**0) + L0x7fffffffb082*(x**1) + L0x7fffffffb084*(x**2) +
:
 L0x7fffffffb17c*(x**126) + L0x7fffffffb17e*(x**127))
[3329, x**128 - (1600)]]
&&
and [
(-6658)@16 <s L0x7fffffffaf80, L0x7fffffffaf80 <s (6658)@16,
:
(-6658)@16 <s L0x7fffffffb17e, L0x7fffffffb17e <s (6658)@16];

(*********** LEVELS 1..6, explained below ***********)
:
```

The incomplete NTT in the AVX2 implementation from PQClean [PQC21] does the following map (where ζ_i denote all the primitive 256th roots of unity in $\mathbb{Z}_q$):

$$\begin{aligned}\mathbb{Z}_q[X]/\langle X^{256}+1\rangle &\to \mathbb{Z}_q[X]/\langle X^{128}-\omega_4\rangle \times \mathbb{Z}_q[X]/\langle X^{128}+\omega_4\rangle \\ \to \cdots &\to \mathbb{Z}_q[X]/\langle X^{2}-\zeta_0\rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^{2}-\zeta_{127}\rangle.\end{aligned}$$

In this AVX2 implementation, a 256-bit SIMD register contains 16 16-bit signed integer coefficients. NTT multipliers (roots of unity) moreover are in Montgomery form. Each multiplication is hence always combined with a signed Montgomery reduction. Because of Montgomery reductions and the small magnitude of q, all coefficients are representable in 16 bits at all seven levels. There are no overflows. No extra reduction is needed.

However, one notes that the AVX2 implementation does not compute NTT strictly by levels. There is level 0, in which all 256 coefficients are used together. Then from level 1 onward, at most 128 coefficients are needed at a time. The implementation therefore uses eight 256-bit SIMD registers to hold the coefficients of the NTT at each level. After level 6 for the first 128 coefficients is done, the last 128 coefficients are loaded and then the NTT levels 1 through 6 for these coefficients are performed.

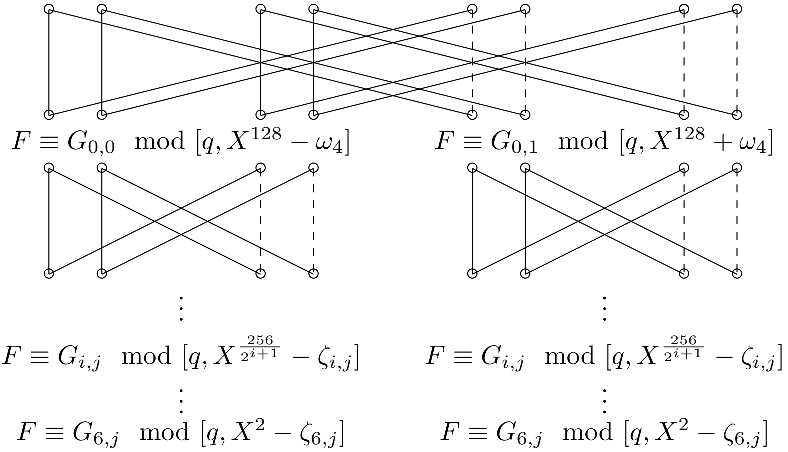


Figure A.8: Workflow of verifying AVX2 implementation for Kyber NTT.

We follow the same strategy in verification. The mid-conditions that we see at the end of level 0 specify that

$$F \equiv G_{0,j} \mod [q, X^{128} - \zeta_{0,j}] \text{ for all } 0 \leq j < 2$$

and

$$-2q < g_{0,j,k} < 2q \text{ for all } 0 \leq j < 2, 0 \leq k < 128.$$

Here the $\zeta_{0,j}$ are 1729 and 1600, the principal 4th roots of unity (denoted as $\pm\omega_4$ in the map above). At level $i > 1$, we specify these mid-conditions for the first 128 coefficients

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}] \text{ for all } 0 \leq j < 2^i$$

and

$$-(2+i)q < g_{i,j,k} < (2+i)q \text{ for all } 0 \leq j < 2^i, 0 \leq k < 256/2^{i+1}.$$

We show the cut at level 1 as an example, first half of coefficients:

```
:
(*********** SUMMARY OF LEVEL 1 0 ***********)

cut
and [
eqmod (inp_poly**2)
(ymm3_0*(x**0) + ymm3_1*(x**1) + ymm3_2*(x**2) + ymm3_3*(x**3) +
:
 ymm6_c*(x**60) + ymm6_d*(x**61) + ymm6_e*(x**62) + ymm6_f*(x**63))
[3329, x**64 - (2580)],
eqmod (inp_poly**2)
(ymm8_0*(x**0) + ymm8_1*(x**1) + ymm8_2*(x**2) + ymm8_3*(x**3) +
:
 ymm11_c*(x**60) + ymm11_d*(x**61) + ymm11_e*(x**62) + ymm11_f*(x**63))
[3329, x**64 - (749)]]
&&
and [
(-9987)@16 <s ymm3_0, ymm3_0 <s (9987)@16,
:
(-9987)@16 <s ymm11_f, ymm11_f <s (9987)@16];
```

The 128 coefficients at the end of the first half level 1 form two degree-63 polynomials related to the input polynomial by modular equivalence. At the same time, each coefficient is guaranteed to be less than $3q$ in magnitude. As shown above, at level i, the polynomials are split further with equivalence modulo various $X^{256/2^{i+1}} - \zeta_{i,j}$ and bound by $\pm(2+i)q$. Similarly, the following mid-conditions are used for the last 128 coefficients at level $i > 1$:

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}] \text{ for all } 2^i \leq j < 2^{i+1}$$

with ranges

$$-(2+i)q < g_{i,j,k} < (2+i)q \text{ for all } 2^i \leq j < 2^{i+1}, 0 \leq k < 256/2^{i+1}.$$

Figure A.8 is an illustration. At the end of the second half of level 6, we fill in the following concluding conditions:

```
#retq                                       #! 0x55555556f751 = 0x55555556f751;

{
and [
eqmod (inp_poly**2)
(L0x7fffffffaf80 + L0x7fffffffafa0*x) [3329, x**2 - (17)],
eqmod (inp_poly**2)
(L0x7fffffffafc0 + L0x7fffffffafe0*x) [3329, x**2 - (3312)],
...
(L0x7fffffffb15e + L0x7fffffffb17e*x) [3329, x**2 - (1175)]]
prove with 6 &&
and [
(-26632)@16 <s L0x7fffffffaf80, L0x7fffffffaf80 <s (26632)@16,
...
```

```
(-26632)@16 <s L0x7fffffffb17e, L0x7fffffffb17e <s (26632)@16]
prove with 6
}
```

The range portion of the ending condition says that every output limb is supposed to be between ±$8q(= 26632)$. The algebraic portion of the ending condition says that every two output coefficients make up a linear polynomial equal to the remainder of the entering polynomial modulo $X^2 - \zeta_i$, with each ζ_i an appropriate root of unity. The "`prove with`" is another compositional reasoning gadget (also see Section A.2.3.2). We do not need any of the shorthands that express integers formed of multiple words and their arithmetic operations and algebraic relations in CryptoLine as they are not used here.

Finally we can run CryptoLine. It obtains from the starting conditions and each CryptoLine instruction corresponding algebraic relations, then verifies each *safety conditions* using the SMT solver, and attempts to deduce the conclusions from the premises. It does so by expressing each algebraic relation as an element in a polynomial ring (one which should be zero when the relation holds). The algebraic part of the conclusions is also converted into polynomial ring elements, and a CAS reduces the ring element representing the conclusion using the ideal spanned by our collection of relations. If the reduction results in zero, then the verification is successful.

```
$ cv -v -isafety -jobs 24 -slicing -no_carry_constraint PQCLEAN_KYBER768_AVX2_polyvec_ntt.cl
Parsing Cryptoline file:                [OK]            0.089273 seconds
Checking well-formedness:               [OK]            0.031599 seconds
Transforming to SSA form:               [OK]            0.019121 seconds
Rewriting assignments:                  [OK]            0.020577 seconds
Verifying program safety:               [OK]            183.994889 seconds
Verifying range assertions:             [OK]            42.385435 seconds
Verifying range specification:          [OK]            200.594131 seconds
Rewriting value-preserved casting:      [OK]            0.001421 seconds
Verifying algebraic assertions:         [OK]            0.007455 seconds
Verifying algebraic specification:      [OK]            26.648724 seconds
Verification result:                    [OK]            453.802915 seconds
```

As shown in the depiction above, the verification has succeeded.

The inverse NTT. The inverse NTT AVX2 implementation for Kyber is symmetric to the description above. The first 128 coefficients are first computed in inverse levels 6 to 1. The computation for the last 128 coefficients then follows. Finally, all 256 coefficients are computed in the inverse level 0. In Kyber inverse NTT, extra Montgomery reductions are needed to make coefficients representable in 16 bits to avoid overflows and underflows. Let $P_j = p_{j,0} + p_{j,1}X$ for $0 \leq j < 128$ be the 128 input polynomials for the inverse NTT.

We have the following

$$-q < p_{j,k} < q \text{ for all } 0 \leq j < 128, 0 \leq k < 2.$$

We specify the following mid-conditions at inverse level i for $6 \geq i > 0$, $0 \leq j < 127$:

$$H_{i,\lfloor j/2^{7-i} \rfloor} \equiv 2^{16-i} P_j \mod [q, X - \zeta_j].$$

Similarly, the mid-conditions for the last 128 coefficients are the same except for $128 \leq j < 256$. Finally, Kyber inverse NTT has the following post-conditions

$$\overline{F} = H_{0,0} \equiv 2^{16} P_j \mod [q, X - \zeta_j]$$

and $-8q < h_{0,0,k} < 8q$ for $0 \leq k < 256$. *Note that the output polynomial has an extra factor of* 2^{16} *after the Kyber inverse NTT because the point multiplication uses Montgomery multiplication, introducing an extra factor of* 2^{-16} *that needs balancing out.*

A.2.3.4 Differences on the Cortex-M4

The ARM Cortex-M4 is a micro-controller usually without an OS to run, so we must use a PC connected to a development kit (here, a STM32F429I-disc1). The `itrace.py` has support for Cortex-M4 that uses `gdb-multiarch`, the multi-architectural gdb. The translation rules differ, but it is otherwise the same process.

The verifier will examine, possibly with the help of the programmer, the actual instructions and possibly inserts subsidiary conditions to prove at various points in the code (cuts, asserts and assumes). A very common occurrence in the examples we deal with is that during Montgomery reduction, a word is asserted and then proved to be congruent to zero mod the radix R, but because R is precisely the word size, we can then assume that this is exactly zero.

A.2.4 Verifying AVX2 Saber implementation

Recall that Saber uses a module of dimension $\ell \times \ell$ over the ring $R_q = \mathbb{Z}_q[x]/\langle X^n + 1\rangle$ with $q = 2^{13}$ and $n = 256$. For performing only a single polynomial multiplication it is usually advantageous to use an incomplete NTT but for Saber wherein the matrix-vector product the vector of polynomials only needs to be transformed once and the inner products can be computed in the NTT basis, a complete NTT is preferable.

The Intel AVX2 implementation uses prime moduli $q_0 = 7681$ and $q_1 = 10753$ for the NTTs of length 256 and maps:

$$R_{q_s} = \mathbb{Z}_{q_s}[X]/\langle X^{256}+1\rangle \to \mathbb{Z}_{q_s}[X]/\langle X-\zeta_0\rangle \times \cdots \times \mathbb{Z}_{q_s}[X]/\langle X-\zeta_{255}\rangle$$

where $q_s \in \{q_0, q_1\}$. A polynomial multiplication in R_q is performed by the following steps. First, the implementation applies two complete size-256 NTTs over $\mathbb{Z}_{q_0}$ to the input polynomials, performs coefficient-wise multiplication, and then applies an inverse NTT over $\mathbb{Z}_{q_0}$. Second, the first step is repeated once but this time all operations are over $\mathbb{Z}_{q_1}$. Finally, the Chinese remainder theorem (CRT) is applied to obtain the multiplication result over $\mathbb{Z}_{q_0 q_1}$.

A.2.4.1 Forward NTT

The input of the NTT routine is a size-256 polynomial with each coefficient ranging between $\pm q/2$. Let $F(X) = \sum_{k=0}^{255} f_k X^k \in \mathbb{Z}_q[X]$ be the input polynomial. We specify the following range pre-conditions

$$-q/2 \le f_k < q/2, \text{ for all } 0 \le k < 256$$

where the algebraic pre-condition is simply `true`.

The NTT routine first performs three levels (levels 0, 1, and 2) of CT butterflies, and then twists all the polynomials. This is followed by another three levels (levels 3, 4, and 5) of CT butterflies, and then all polynomials are twisted again. Finally, two additional CT butterflies (levels 6 and 7) are performed. Extra Montgomery reductions are applied when needed to make coefficients representable in 16 bits to avoid overflows and underflows. We detail the post-conditions and the mid-conditions in the following paragraphs.

Let $G_{i,j}(X) = \sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k} X^k \in \mathbb{Z}_{q_s}[X]$ be the polynomials at the end of level i for $0 \le i \le 7$ and $0 \le j < 2^{i+1}$. Let $\zeta_{i,j}$ be the roots of unity in $\mathbb{Z}_{q_s}$ at level i with $0 \le i \le 7$ and $0 \le j < 2^{i+1}$. The first three levels map

$$\begin{aligned}
\mathbb{Z}_{q_s}[X]/\langle X^{256}+1\rangle &\to \prod_{j=0}^{1} \mathbb{Z}_{q_s}[X]/\langle X^{128}-\zeta_{0,j}\rangle \\
&\to \prod_{j=0}^{3} \mathbb{Z}_{q_s}[X]/\langle X^{64}-\zeta_{1,j}\rangle \\
&\to \prod_{j=0}^{7} \mathbb{Z}_{q_s}[X]/\langle X^{32}-\zeta_{2,j}\rangle.
\end{aligned}$$

At the end of level i for $0 \leq i \leq 2$, we specify the algebraic mid-conditions for $0 \leq j < 2^{i+1}$:

$$F(X) \equiv G_{i,j}(X) \mod [q_s, X^{256/2^{i+1}} - \zeta_{i,j}].$$

Polynomials $G_{2,j}(X)$ are then twisted before the next three levels of CT butterflies.

Consider a polynomial $G_{2,j}(X) \in \mathbb{Z}_{q_s}[X]/\langle X^{32} - \zeta_{2,j}\rangle$ at the end of level 2. Let α_j be a 32nd root of $\zeta_{2,j}$. The polynomial is twisted by multiplying each coefficient $g_{2,j,k}$ with α_j^k based on the following mapping:

$$\mathbb{Z}_{q_s}[X]/\langle X^{32} - \zeta_{2,j}\rangle \quad \to \quad \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{32} - 1\rangle$$

with $X = \alpha_j Y_j$. Define

$$\zeta'_{i,j} = \begin{cases} 1 & \text{if } j = 0, \\ -1 & \text{if } j = 1, \\ \zeta_{i-1,j-2} & \text{if } i \geq 1 \text{ and } j \geq 2. \end{cases}$$

The level of CT butterflies in level 3 after twisting is based on the following mappings:

$$\mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{32} - 1\rangle \quad \to \quad \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle$$

where $0 \leq j < 8$. Thus we have

$$F(X) \equiv G_{3,j}(Y) \mod [q_s, X - \alpha_j Y_j, Y_j^{16} - \zeta'_{0,j \bmod 2}]$$

for $0 \leq j < 16$ at the end of level 3. Polynomials over Y can be rewritten as polynomials over X based on the following mappings:

$$\begin{aligned} & \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle \\ \to \quad & \mathbb{Z}_{q_s}[X]/\langle X^{16} - \alpha_j^{16}\zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[X]/\langle X^{16} - \alpha_j^{16}\zeta'_{0,1}\rangle. \end{aligned}$$

We have the algebraic mid-conditions at the end of level 3 for $0 \leq j < 16$:

$$F(X) \equiv G_{3,j}(\alpha_{\lfloor j/2 \rfloor}^{-1} X) \mod [q_s, X^{16} - \alpha_{\lfloor j/2 \rfloor}^{16} \zeta'_{0,j \bmod 2}].$$

Level 4 is based on the following mappings:

$$\mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,0}\rangle \times \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{16} - \zeta'_{0,1}\rangle \quad \to \quad \prod_{t=0}^{3} \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^{8} - \zeta'_{1,t}\rangle$$

where $0 \le j < 8$. Again, polynomials over Y at the end of level 4 can be rewritten as polynomials over X based on the following mappings:

$$\prod_{t=0}^{3} \mathbb{Z}_{q_s}[Y_j]/\langle Y_j^8 - \zeta'_{1,t}\rangle \quad \rightarrow \quad \prod_{t=0}^{3} \mathbb{Z}_{q_s}[X]/\langle X^8 - \alpha_j^8 \zeta'_{1,t}\rangle.$$

We have the algebraic mid-conditions at the end of level 4 for $0 \le j < 32$:

$$F(X) \equiv G_{4,j}(\alpha^{-1}_{\lfloor j/4 \rfloor} X) \mod [q_s, X^8 - \alpha^8_{\lfloor j/4 \rfloor} \zeta'_{1,j \bmod 4}].$$

The CT butterfly in Level 5 is applied in the same way. In general, at the end of level i for $3 \le i \le 5$, we specify the algebraic mid-conditions for $0 \le j < 2^{i+1}$:

$$\begin{aligned} F(X) \equiv \quad & G_{i,j}(\alpha^{-1}_{\lfloor j/2^{i-2} \rfloor} X) \\ \bmod \quad & [q_s, X^{256/2^{i+1}} - \alpha^{256/2^{i+1}}_{\lfloor j/2^{i-2} \rfloor} \zeta'_{i-3, j \bmod (2^{i-2})}]. \end{aligned}$$

Polynomials after level 5 are twisted again before the last two levels of CT butterflies. Let β_j be the 4th root of $\zeta'_{2,j \bmod 8}$ for $0 \le j < 64$. Similar to the twisting after level 2, each polynomial $G_{5,j}$ for $0 \le j < 64$ is twisted by multiplying $g_{5,j,k}$ with β_j^k. For $6 \le i \le 7$ and $0 \le j < 2^{i+1}$, we specify the algebraic mid-conditions:

$$\begin{aligned} F(X) \equiv \quad & G_{i,j}(\alpha^{-1}_{\lfloor j/2^{i-2} \rfloor} \beta^{-1}_{\lfloor j/2^{i-5} \rfloor} X) \\ \bmod \quad & [q_s, X^{256/2^{i+1}} - \alpha^{256/2^{i+1}}_{\lfloor j/2^{i-2} \rfloor} \beta^{256/2^{i+1}}_{\lfloor j/2^{i-5} \rfloor} \zeta'_{i-6, j \bmod 2^{i-5}}]. \end{aligned}$$

Specifically the algebraic mid-conditions after level 7 are

$$F(X) \equiv G_{7,j}(\alpha^{-1}_{\lfloor j/32 \rfloor} \beta^{-1}_{\lfloor j/4 \rfloor} X) \mod [q_s, X - \alpha_{\lfloor j/32 \rfloor} \beta_{\lfloor j/4 \rfloor} \zeta'_{1,j \bmod 4}]$$

for $0 \le j < 256$. Define $\zeta_j = \alpha_{\lfloor j/32 \rfloor} \beta_{\lfloor j/4 \rfloor} \zeta'_{1,j \bmod 4}$. The algebraic post-conditions specified are:

$$F(X) \equiv G_{7,j}(X) \mod [q_s, X - \zeta_j]$$

for $0 \le j < 256$.

The ranges of coefficients do not simply increase by q after each CT butterfly because of twisting polynomials after levels 2 and 5 and extra Montgomery reductions. Instead, ranges of coefficients are computed by the program `test_range256n` from the `ntt-polymul` repository (see Footnote 2) and are asserted in the mid-conditions after each CT butterfly.

CryptoLine successfully verifies all the mid-conditions and the post-conditions specified for the NTT routine.

A.2.4.2 Inverse NTT

The inverse NTT Intel AVX2 implementation for Saber is symmetric. It first computes two layers of GS butterflies in inverse levels 7 to 6 followed by a twisting (at the end of level 6), and then three layers of GS butterflies in inverse levels 5 to 3 followed by another twisting (at the end of level 3). Finally, three layers of GS butterflies are computed in levels 2 to 0. Let $P_j = p_j$ for $0 \leq j < 256$ be the 256 input polynomials for the inverse NTT. The algebraic pre-condition for the inverse NTT routine is simply `true`. Let $H_{i,j}(X) = \sum_{k=0}^{256/2^i-1} h_{i,j,k}X^k$ be the polynomials obtained at the end of inverse level i for $7 \geq i \geq 0$. We specify the mid-conditions at the end of inverse level 7:

$$2P_j \equiv H_{7,j}(\alpha^{-1}_{\lfloor j/32 \rfloor}\beta^{-1}_{\lfloor j/4 \rfloor}X) \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$. Inverse level 6 contains one layer of GS butterflies followed a twisting. At the end of inverse level i for $6 \geq i \geq 4$, the mid-conditions are:

$$2^{8-i}P_j \equiv H_{i,j}(\alpha^{-1}_{\lfloor j/32 \rfloor}X) \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$. Inverse level 3 contains one layer GS butterfly followed by another twisting. At the end of level i for $3 \geq i \geq 0$, the mid-conditions are:

$$2^{8-i}P_j \equiv H_{i,j}(X) \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$. Define $\overline{F} = H_{0,0}$. The algebraic post-conditions of the inverse NTT routine are then:

$$\overline{F} \equiv 2^{8-i}P_j \mod [q_s, X - \zeta_j]$$

for $0 \leq j < 256$.

To speed up verification, algebraic mid-conditions at the ends of inverse levels 7 to 4 are actually removed because the algebraic mid-conditions at the end of inverse level 3 can be easily verified without any preceding mid-condition. Moreover, we apply the non-local compositional reasoning technique in inverse levels 3 to 0. Consider for example an inverse level i for $2 \geq i \geq 0$. Every modular equation at the end of inverse level i is only related to one modular equation at the end of inverse level $i + 1$. However, the mid-conditions at the end of inverse level $i + 1$ are specified in a single cut and thus, all of them are taken into account by computer algebra systems when proving a modular equation at the end of inverse level i. To improve performance, following the mid-conditions at the end of inverse level $i + 1$, we add one `cut` for each modular equation appearing in the mid-conditions. We are then able add one prove with to refer to the only one related modular equation in inverse level $i+1$ for each modular equation to be verified in the mid-conditions at the end of inverse level i. Therefore hundreds of modular equations are eliminated from the problems submitted to computer algebra systems. Such application of non-local compositional reasoning drastically reduces the verification time.

The ranges of coefficients are computed by the program `test_range256n` and are asserted in the range mid-conditions after each GS butterflies.

CryptoLine verifies all the mid-conditions and the post-conditions except the range mid-conditions at the end of level 6. This failure is due to a mismatch of the extra reduction in level 6. The implementation of inverse NTT applies one extra reduction at the end of level 6 while the programmer's own range computation program `test_range256n` applies the extra reduction at the beginning of level 7. After the fix of the range computation program, we specify new ranges at the end of level 6 and CryptoLine verifies all the mid-conditions and the post-conditions. Note that the program was correct; it was the programmer's range-checker that was wrong. As a result of our work, the range-checking tool was fixed in commit https://github.com/ntt-polymul/ntt-polymul/commit/7d88aa6b051bd076cc054eafd257c4ae8c10617c.

A.2.5 Verifying Cortex-M4 `ntruhps2048509` Implementation

The `ntruhps2048509` M4 implementation leverages the following mapping, where ζ_i denote all the 256th roots of unity in $\mathbb{Z}_{q'}$ with $q' = 1043969$:

$$\mathbb{Z}_{q'}[X]/\langle X^{1024}+1\rangle \to \mathbb{Z}_{q'}[X]/\langle X^4-\zeta_0\rangle \times \cdots \times \mathbb{Z}_{q'}[X]/\langle X^4-\zeta_{255}\rangle.$$

The implementation first transforms the polynomial via incomplete size-1024 NTT comprising 2 sets of 4-layer NTTs (CT butterflies), and performs each 4-coefficient multiplication (modulo a degree-3 polynomial) with schoolbook. Then it does 2 sets of 3-layer inverse NTTs (GS butterflies), followed by a final stage. The final stage consists of the following operations: 2 layers of inverse NTTs, taking $\mathrm{mod}(X^{509}-1)$, Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_{\mathsf{N}}^{-1} \bmod q'$, and reducing coefficients to the ring $\mathbb{Z}_q$ [CHK+21]. The constants $\mathtt{R}$ and $\mathsf{NTT}_{\mathsf{N}}$ are 2^{32} and 256, respectively. Coefficients in the implementation use the signed 32-bit representation.

A.2.5.1 Forward NTT

The input of the NTT routine is a degree-508 polynomial with each coefficient ranging between $\pm q$. Let $F = \sum_{k=0}^{508} f_k X^k$ be the input polynomial. The pre-conditions are

$$-q \leq f_k < q, \text{ for all } 0 \leq k \leq 508.$$

The implementation first performs central reduction for each coefficient to normalize the range between $\pm q/2$ before NTT. The 8-level NTT is then computed in two phases: 4-layer NTTs from level 0 to level 3 are calculated first, followed by 4-layer NTTs from level 4 to level 7.

Define polynomial $\hat{F} = \sum_{k=0}^{508} \hat{f}_k X^k$ to be the result of the central reduction. Let $G_{i,j} = \sum_{k=0}^{1024/2^{i+1}-1} g_{i,j,k} X^k$ denote the polynomials obtained at the end of level i, where $0 \leq i \leq 7$, $0 \leq j < 2^{i+1}$, and $\zeta_{i,j}$ with $0 \leq j < 2^{i+1}$ the roots of unity at level i. The output polynomials of the NTT routine are therefore $G_{7,j} = \sum_{k=0}^{3} g_{7,j,k} X^k$ with $0 \leq j < 256$. The post-conditions to be verified are

$$F \equiv \hat{F} \mod q \qquad \text{and} \qquad \hat{F} \equiv G_{7,j} \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256$$

with ranges

$$-128q' < g_{7,j,k} < 128q', \text{ for all } 0 \leq j < 256, 0 \leq k < 4.$$

The first part of the algebraic post-conditions represents the correctness of the central reduction, while the second part specifies the correctness of the 8-level NTT. CryptoLine successfully verifies the correctness of the NTT routine with respect to the aforementioned pre- and post-conditions.

In order to improve the verification efficiency, we further utilize the compositional reasoning mechanism provided by the `cut` instruction. As the 8-level NTT is clearly demarcated into two phases, we specify the following mid-conditions at the end of the first phase (level 3) to split the whole verification problem into two smaller sub-problems:

$$F \equiv \hat{F} \mod q \qquad \text{and} \qquad \hat{F} \equiv G_{3,j} \mod [q', X^{64} - \zeta_{3,j}], \text{ for all } 0 \leq j < 16 \tag{A.1}$$

with ranges

$$-5q' < g_{3,j,k} < 5q', \text{ for all } 0 \leq j < 16, 0 \leq k < 64.$$

In fact, we divide the sub-problems into even smaller pieces to achieve more efficiency, thanks to the non-local compositional reasoning feature supported by the `cut` instruction. For example in the first phase, the 4-layer NTTs are performed iteratively. Each iteration only transforms 16 coefficients. We thus insert the following mid-conditions at the end of the eth iteration for $0 \leq e < 64$ to specify the computation of that iteration:

$$f_{64k+e} \equiv \hat{f}_{64k+e} \mod q, \text{ for all } 0 \leq k < 16 \tag{A.2}$$

and

$$g_{3,j,e}X^e \equiv \sum_{k=0}^{15} \hat{f}_{64k+e}X^{64k+e} \mod [q', X^{64} - \zeta_{3,j}], \text{ for all } 0 \leq j < 16 \tag{A.3}$$

where we assume $f_k = \hat{f}_k = 0$ when $k > 508$. We use the "`prove with`" extension of `cut` in the mid-conditions A.1 to add mid-conditions A.2 and A.3 as extra premises to ease the verification.

A.2.5.2 Inverse NTT

The inverse NTT routine consists of three phases. Phases I and II transform all 1024 coefficients by 3-layer GS butterflies from inverse levels 7 to 5 and from inverse levels 4 to 2, respectively. In phase III, 2-layer inverse NTTs from inverse levels 1 to 0 are performed iteratively, with 4 coefficients at each iteration. In the same iteration, for each resulting coefficient, the mapping $\mathbb{Z}_{q'}[X]/\langle X^{1024}+1\rangle \to \mathbb{Z}_{q'}[X]/\langle X^{509}-1\rangle$ is calculated immediately, followed by Montgomery multiplication by the factor $\mathtt{R}^2\mathsf{NTT}_\mathsf{N}^{-1} \bmod q'$ and finally reduction to the ring $\mathbb{Z}_q$.

To formalize appropriately the post-conditions, we denote several polynomials by the following notations:

- $P_j = \sum_{k=0}^{3} p_{j,k}X^k$ with $0 \leq j < 256$, the input polynomials for the inverse NTT routine;
- $\overline{F} = \sum_{k=0}^{1023} \overline{f}_k X^k$, the polynomial obtained at the end of 8-level inverse NTT;
- $F^* = \sum_{k=0}^{508} f_k^* X^k$, the remainder polynomial after taking $\mathrm{mod}(X^{509}-1)$ and Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_\mathsf{N}^{-1} \bmod q'$ in phase III;
- $\tilde{F} = \sum_{k=0}^{508} \tilde{f}_k X^k$, the output polynomial of the inverse NTT routine.

The post-conditions to be verified are therefore specified as follows:

$$\overline{F} \equiv 256P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256 \tag{A.4}$$

$$\mathsf{NTT}_\mathsf{N}F^* \equiv \mathtt{R}\overline{F} \mod [q', X^{509} - 1] \tag{A.5}$$

$$\tilde{F} \equiv F^* \mod q \tag{A.6}$$

with appropriate ranges. Condition A.4 constitute the correctness of the 8-level inverse NTT, while condition A.5 ensures the correctness of both the modulo operation by $X^{509}-1$ and Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_\mathsf{N}^{-1} \bmod q'$ in phase III. Condition A.6 means the final reduction is correct.

Similarly, we construct mid-conditions to make the verification more efficient, thanks to the clear three-phase structure of the implementation. Let $H_{i,j} = \sum_{k=0}^{1024/2^i-1} h_{i,j,k}X^k$ be the polynomials obtained at the end of inverse level i with $7 \geq i \geq 0$ and $0 \leq j < 2^i$. Then we have the following mid-conditions at the end of phase I (inverse level 5)

$$H_{5,\lfloor j/8 \rfloor} \equiv 2^3 P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256;$$

and the following mid-conditions at the end of phase II (inverse level 2)

$$H_{2,\lfloor j/64 \rfloor} \equiv 2^6 P_j \mod [q', X^4 - \zeta_{7,j}], \text{ for all } 0 \leq j < 256.$$

Moreover, we define more refined mid-conditions in a similar way to the verification of the NTT routine, since the phases in the inverse NTT routine are also implemented by iterations. We refer the interested readers to the supplementary material for the detailed mid-conditions that have been used.

Interestingly, when verifying the post-condition A.5, CryptoLine reports "failed". This post-condition corresponds to the correctness of both the modulo operation by $X^{509}-1$ and Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_\mathsf{N}^{-1} \bmod q'$ in phase III. The failure indicates either that these computations are flawed, or that the computations are correct yet CryptoLine is not able to verify with existing premises. After inspecting the error and related code, we found that the following modular equation can be verified with CryptoLine:

$$\mathsf{NTT}_\mathsf{N} f_2^* \equiv \mathtt{R}(\overline{f}_2 + \overline{f}_{511} + \overline{f}_{1017}) \mod [q'].$$

Nevertheless, note that condition A.5 requires the following modular equation:

$$\mathsf{NTT}_\mathsf{N} f_2^* X^2 \equiv \mathtt{R}(\overline{f}_2 X^2 + \overline{f}_{511} X^{511} + \overline{f}_{1020} X^{1020}) \mod [q', X^{509}-1].$$

Since $X^{1017} \not\equiv X^2 \mod [X^{509}-1]$, the code does not appear to calculate the coefficient f_2^* correctly. Yet the problem evades all test inputs. There must be a simple explanation.

It turns out that we need additional premises verify post-condition A.5. Recall that the inverse NTT routine is only for `ntruhps2048509`. As a part of NTT multiplication between two degree-508 polynomials, the modulo operation by $X^{509}-1$ in phase III will take as input a polynomial of degree less than 1017. Thus $\overline{f}_k = 0$ for $1017 \leq k \leq 1023$ in this context. Since $\overline{f}_{1020} = \overline{f}_{1017} = 0$, the routine is correct only if it is used in `ntruhps2048509`. With the observation, we add these assumptions with the following instructions:

$$\mathsf{assume}\ \overline{f}_k = 0\ \&\&\ \mathsf{true}, \text{ for all } 1017 \leq k \leq 1023.$$

Then CryptoLine successfully verifies all the post-conditions. These `assume`'s illustrate that the inverse NTT routine in question, in particular the modulo operation by $X^{509} - 1$ in phase III, is not generally correct. It is correct when being a part of NTT multiplication between two degree-508 polynomials. CryptoLine forces the verifier to specify precisely all the premises required to show correctness, hence helps the programmer and the users to better understand both the generality and limitations of the code.

A.2.6 Other implementations

As aforementioned in Section A.2.1.1, verification has been carried out on six chosen NTT implementations. We have explained the details on how to verify the three of them, including the AVX2 NTT implementations for Kyber and Saber, and the Cortex-M4 NTT implementation for NTRU. Although the remaining implementations are optimized differently, they are built with basic blocks such as CT/GS butterflies, twisting and Montgomery reductions that we have seen already. The techniques to construct the verification conditions are similar. We demonstrate briefly the primary verification conditions for them in the following. The details of all the conditions employed can be found in our supplementary material.

A.2.6.1 Cortex-M4 `kyber768` implementation

The Kyber M4 implementation from `pqm4`[3] maps

$$\mathbb{Z}_q[X]/\langle X^{256} + 1\rangle \to \mathbb{Z}_q[X]/\langle X^2 - \zeta_0\rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^2 - \zeta_{127}\rangle$$

with ζ_j for the primitive 256th roots of unity. The 7-level NTT is structured with 2 sets of 3-layer CT butterflies and then a set of 1-layer CT butterflies followed by Barrett reductions. The inverse NTT is symmetric with GS butterflies.

For the forward NTT routine, let $F = \sum_{k=0}^{255} f_k X^k$ be the input polynomial, $G_{i,j} = \sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k} X^k$ the polynomials at the end of level i with $0 \leq i \leq 6$ and $0 \leq j < 2^{i+1}$, and $\zeta_{i,j}$ be the roots of the unity at the end of level i. As for the inverse, $P_j = p_{j,0} + p_{j,1}X$ $(0 \leq j < 128)$ denote the 128 input polynomials, and $H_{i,j} = \sum_{k=0}^{256/2^{i}-1} h_{i,j,k} X^k$ for the polynomials at the end of inverse level i with $6 \geq i \geq 0$ and $0 \leq j < 2^{i+1}$, where the output polynomial $\overline{F} = H_{0,0}$.

The range pre-condition $-q \leq f_k < q$ is used for each coefficient f_k with $0 \leq k < 256$ when verifying the NTT routine. We specify the following algebraic mid-conditions at the end of levels $i = 2$ and $i = 5$:

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}], \text{ for all } 0 \leq j < 2^i.$$

The above equations with $i = 6$ are the algebraic post-conditions to be verified. On the other hand, the range post-conditions are $0 \leq g_{6,j,k} \leq q$ due to Barrett reductions.

For the inverse NTT, the algebraic mid-conditions inserted at the end of inverse levels $i = 6$ and $i = 3$ become

$$H_{i,\lfloor j/2^{7-i} \rfloor} \equiv 2^{7-i} P_j \mod [q, X^2 - \zeta_j], \text{ for all } 0 \leq j < 128.$$

Finally, the algebraic post-conditions at the end of inverse level 0 are

$$\overline{F} \equiv 2^{16} P_j \mod [q, X^2 - \zeta_j], \text{ for all } 0 \leq j < 128$$

with an extra factor 2^9 being the effect of Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_{\mathsf{N}}^{-1}$. The range mid-conditions and post-conditions are all $-q \leq h_{i,j,k} < q$ for each coefficient $h_{i,j,k}$.

A.2.6.2 Cortex-M4 Saber implementation

The implementation from [ACC+21] maps

$$\mathbb{Z}_q[X]/\langle X^{256} + 1\rangle \to \mathbb{Z}_q[X]/\langle X^4 - \zeta_0\rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^4 - \zeta_{63}\rangle.$$

Thus the NTT routine performs 2 sets of 3-layer NTTs via CT butterflies. To use CT butterflies as well in the inverse NTT, the mapping is rewritten as follows:

$$\begin{array}{rcl} \mathbb{Z}_q[X]/\langle X^{256} + 1\rangle & \to & \mathbb{Z}_q[X,Y]/\langle X^4 - Y\rangle\langle Y^{64} + 1\rangle \\ & \overset{Y=\zeta Y_{0,0}}{\to} & \mathbb{Z}_q[X,Y_{0,0}]/\langle X^4 - \zeta Y_{0,0}\rangle\langle Y_{0,0}^{64} - 1\rangle \\ & \vdots & \\ & \to & \prod_{j=0}^{63} \mathbb{Z}_q[X,Y_{6,j}]/\langle X^4 - \zeta_j Y_{6,j}\rangle\langle Y_{6,j} - 1\rangle \\ & \to & \prod_{j=0}^{63} \mathbb{Z}_q[X]/\langle X^4 - \zeta_j\rangle \end{array}$$

where $Y_{i,j}$ are the fresh variables introduced by the ith twisting. The twisted inverse NTT routine therefore consists of 2 sets of 3-layer CT butterflies, followed by a twisting mixed with Montgomery multiplication by $\mathtt{R}^2\mathsf{NTT}_{\mathsf{N}}^{-1}$, and a central reduction at last.

For the forward NTT, let $F = \sum_{k=0}^{255} f_k X^k$ again be the input polynomial, $G_{i,j} = \sum_{k=0}^{256/2^{i+1}-1} g_{i,j,k} X^k$ the polynomials at the end of level i for $0 \leq i \leq 5$ and $0 \leq j < 2^{i+1}$. For the inverse routine, define $P_j = \sum_{k=0}^{3} p_{j,k} X^k$ $(0 \leq j < 64)$ as the 64 input polynomials, $H_{i,j} = \sum_{k=0}^{256/2^i-1} h_{i,j,k} X^{(k \bmod 4)} Y_{i,j}^{\lfloor k/4 \rfloor}$ with $0 \leq j < 2^i$ the polynomials obtained at the end of inverse level i $(5 \geq i \geq 0)$, and $\overline{F} = \sum_{k=0}^{255} \overline{f}_k X^k$ the output polynomial of the inverse NTT routine.

As a standard NTT, the NTT routine should satisfy

$$F \equiv G_{i,j} \mod [q, X^{256/2^{i+1}} - \zeta_{i,j}], \text{ for } 0 \leq j < 2^{i+1} \tag{A.7}$$

and

$$-(i+2)q \leq g_{i,j,k} < (i+2)q, \tag{A.8}$$

at the end of level i. The post-conditions conditions A.7 and A.8 with $i = 5$, and the instances when $i = 2$ are inserted as mid-conditions at the end of level 2 for verification efficiency.

As for the inverse routine, the following mid-conditions are specified at the end of inverse level 3:

$$H_{3,\lfloor j/8 \rfloor} \equiv 2^3 P_j \mod [q, X^4 - \zeta_j Y_{6,j}, Y_{6,j} - 1], \text{ for } 0 \leq j < 64$$

and

$$-8q \leq h_{3,j,k} < 8q.$$

Before post-conditions, the following algebraic mid-conditions are inserted:

$$\overline{F} \equiv \mathtt{R} P_j \mod [q, X^4 - \zeta_j Y_{6,j}, Y_{6,j} - 1], \text{ for all } 0 \leq j < 64.$$

Note that, unlike Section A.2.4, we did not eliminate the variables Y's in the above mid-conditions when dealing the twisting. CryptoLine allows both ways of formulating the conditions. Finally, the algebraic post-conditions are verified:

$$\overline{F} \equiv \mathtt{R} P_j \mod [q, X^4 - \zeta_j], \text{ for all } 0 \leq j < 64,$$

with explicitly instantiating $Y_{6,j}$ with 1 using `assume`'s for $0 \leq j < 64$ before to prove the algebraic post-conditions. Because of central reductions at the end, the range post-conditions are $-q/2 \leq \overline{f}_k < q/2$ for all output coefficients $\overline{f}_k$.

A.2.6.3 AVX2 `ntruhps2048509`

The `ntruhps2048509` AVX2 implementation from [CHK+21] maps

$$\mathbb{Z}_q[X]/\langle X^{1024} - 1 \rangle \to \mathbb{Z}_q[X]/\langle X^2 - \zeta_0 \rangle \times \cdots \times \mathbb{Z}_q[X]/\langle X^2 - \zeta_{511} \rangle,$$

with ζ_i again ranging over all the primitive 512nd roots of unity. Both the 9-level NTT and inverse NTT are implemented layer by layer.

As usual, for the forward NTT routine, we use $F = \sum_{k=0}^{511} f_k X^k$ to denote the input polynomial of degree 511, $G_{i,j} = \sum_{k=0}^{1024/2^{i+1}-1} g_{i,j,k} X^k$ for the polynomial obtained at the end of level i with $0 \leq i \leq 8$ and $0 \leq j < 2^{i+1}$, and $\zeta_{i,j}$ with $0 \leq j < 2^{i+1}$ for the roots of unity at level i. As for the inverse NTT routine, $P_j = p_{j,0} + p_{j,1}X$ $(0 \leq j < 512)$ represent the input polynomials, $H_{i,j} = \sum_{k=0}^{1024/2^i-1} h_{i,j,k} X^k$ the resulting polynomials at the end of inverse level i with $8 \geq i \geq 0$ and $0 \leq j < 2^i$, and $\overline{F} = \sum_{k=0}^{1023} \bar{f}_k X^k$ for the output polynomial. In this implementation, $\overline{F} = H_{0,0}$.

As a standard NTT, the NTT routine should satisfy

$$F \equiv G_{i,j} \mod [q, X^{1024/2^{i+1}} - \zeta_{i,j}], \text{ for all } 0 \leq j < 2^{i+1} \tag{A.9}$$

at the end of level i. Thus the algebraic post-conditions to be verified are

$$F \equiv G_{8,j} \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512.$$

On the other hand, the inverse NTT routine should satisfy

$$H_{i,\lfloor j/2^{9-i} \rfloor} \equiv 2^{9-i} P_j \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512 \tag{A.10}$$

at the end of inverse level i. The algebraic post-conditions of the inverse NTT routine are

$$\overline{F} \equiv 2^9 P_j \mod [q, X^2 - \zeta_{8,j}], \text{ for all } 0 \leq j < 512.$$

Range conditions of the coefficients are computed by `test_range1024` from the `ntt-polymul` repository (see Footnote 2).

For efficiency, mid-condition A.9 is only added at the end of level 2 when verifying the NTT routine, while mid-condition A.10 is only inserted at the end of inverse level 3 for the inverse NTT routine. Moreover, since the implementation divides the 1024 coefficients into 8 parts and calculates coefficients in each part separately, more refined mid-conditions are also used to further reduce verification time.

A.2.7 Results

Table A.6: Verification results (in seconds).

architecture	*direction*	*algebra*	*overflow*	*range*	*total*
`kyber768`					
AVX2	normal	26.6	183.9	242.8	453.8
	inverse	761.7	781.0	6050.0	7593.5
Cortex M4	normal	134.3	173.7	191.0	499.4
	inverse	1481.0	348.6	184.1	2014.3
`ntruhps2048509`					
AVX2	normal	478.4	1229.8	1738.6	3447.8
	inverse	3868.6	1545.3	12170.3	17585.7
Cortex M4	normal	1353.0	5970.7	4810.2	12135.2
	inverse	11315.1	3019.6	7813.7	22150.9
`saber`					
AVX2	normal	60.1	207.7	271.7	539.9
	inverse	436.2	443.8	859.4	1741.0
Cortex M4	normal	110.2	2731.9	2196.7	5039.3
	inverse	3250.5	2754.0	853.4	6858.8

We use CryptoLine to verify the AVX2 and Cortex M4 assembly implementations for the NTTs for Kyber, NTRU, and Saber. All experiments are running on an Ubuntu 20.04.3 server with 3.2GHz Intel Xeon and 1TB RAM. Table A.6 shows the verification time for each instance in seconds. In the table, the column *algebra* shows the time for verifying algebraic properties; *overflow* gives the time for checking overflows and underflows; *range* contains the time for range checks; and *total* is the total running time for the instance. All time is in seconds.

Verification time varies drastically among the experiments. Consider, for instance, the experiments of the AVX2 implementation for Kyber NTT. The total verification time for inverse NTT is about 16.7 times slower than those for NTT. From Table A.6, we see that the time for overflow and range checking is drastically different in both instances. In our verification, coefficient ranges are specified and verified for each level in NTT. On the other hand, coefficient ranges are only specified and verified for the inverse levels 1 and 0 in inverse NTT. Range checking is thus divided into 7 sub-tasks in NTT whereas it is divided into two in inverse NTT. Compositional reasoning divides large verification tasks into smaller tasks. In AVX2 Kyber NTT and inverse NTT implementations, we observe significant differences in their verification time.

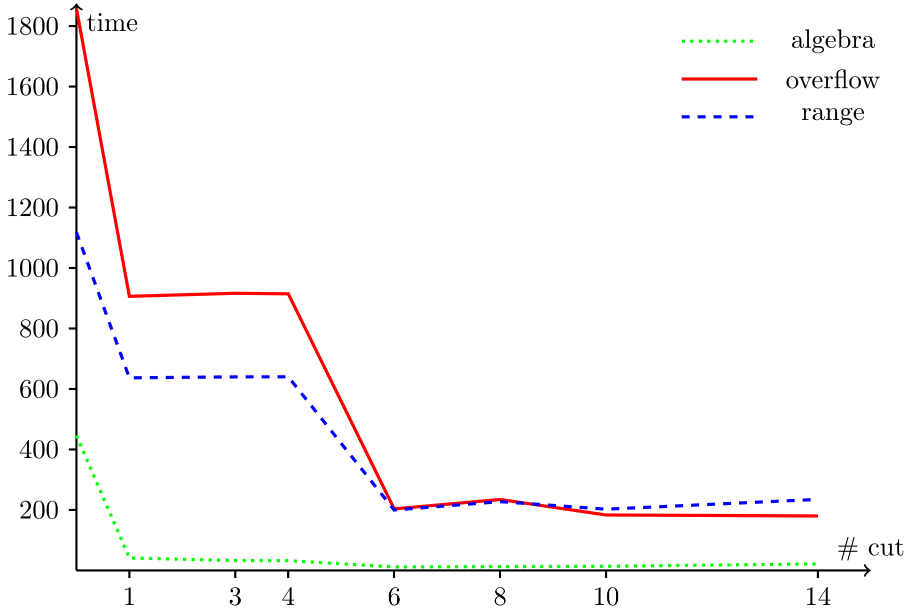


Figure A.9: Effectiveness of Cuts in AVX2 Kyber NTT.

To evaluate the effectiveness of compositional reasoning with cuts, we compare verification time of AVX2 Kyber NTT by numbers of cuts (Figure A.9). The NTT implementation is divided by different numbers of cuts in the figure. The verification time for algebraic properties is drawn in the dotted green line. The time for overflow checking is in the solid red line. And the time for range checks is the dashed blue line. The AVX2 Kyber NTT implementation in Table A.6 uses 14 cuts. It corresponds to the rightmost values in Figure A.9. From the figure, we see the monolithic verification time without cuts is the worst. Adding one cut improves the verification time in all categories significantly. The verification time is similar to one, three, or four cuts in our experiments. However, it improves significantly again with six or more cuts. Most interestingly, the best verification time is with 10 cuts. Adding more cuts in fact increases the verification time slightly.

Figure A.9 shows that compositional reasoning is better than monolithic verification. The verification time can be reduced by 50% with a single cut.

Our experiments also point out limitations of compositional reasoning. First, not all decomposition are effective for verification. Adding more cuts may not improve verification time significantly. Verifiers still need to decide how to divide verification tasks more effectively. Second, extreme decomposition may be harmful. Compositional reasoning necessarily induces overhead. In the extreme case, benefits of compositional reasoning can be nullified by its overhead. Compositional reasoning does not always improve verification time.

Among the three KEM lattice finalists, the verification time of the NTT for `ntruhps2048509` is much longer than the others. This is because it considers input polynomials of size 1024 and performs 10-level NTT. Kyber and Saber have input polynomials of size 256 with 7- and 8-level NTT respectively. In all cases, inverse NTT implementations always take more time to verify. Recall that NTT always has input polynomials of (very) high degrees and output polynomials of degrees 0 or 1. Subsequently, mid-conditions become simpler at each level of NTT computation. At the last level, computer algebra systems only need to verify modular equations over linear or constant polynomials. Inverse NTT however has the opposite pattern. At each level of inverse NTT computation, mid-conditions become more complex. In the end, computer algebra systems need to verify modular equations over high-degree polynomials. The verification time for algebraic properties is much longer in inverse NTT than those in NTT. Differences in overflow or range checking between NTT and inverse NTT are not so pronounced. Rather, they depend more on the number of cuts. For well-decomposed inverse NTT implementations, their overflow or range checking time can be less than corresponding NTT implementations.

We should also mention the effects of our modifications to CryptoLine. Without non-local cutting, it is not possible to cut Kyber at each level because of the structure of the NTT, in which half of the coefficients are used for most layers; as a result variables move in and out of registers. Without ghosts variables (which enable non-local cuts), one can only relate to the last cut. So effectively the only possibility of cutting is somewhere in the code where all variables are written out to memory, which only happens after layers 0 and 6 for Kyber (this is program-dependent). Initially, without the non-local compositional reasoning, we tried verifying Kyber (the smallest of the programs verified) and it took 8× as much time as with the new extension. In NTRU, with its larger state, Singular choked due to the size of the ideal — on a server with 1TB of RAM.

Human time. Perhaps more important than computer clock time is *human time.* Each of our verifications took less than a week of calendar time, and the majority of it was really communication with the programmer of the code, and secondly reading and gaining a basic understanding of the program at hand. We take this opportunity to note that in no case was the verifier the programmer of the code, although in all cases the programmer either provided very good annotations or was cooperative in resolving any questions that arose.

Conclusion. We demonstrate the feasibility for a programmer to verify his or her high-speed assembly code for PQC, as well as for a verification specialist to verify someone else's high-speed PQC software in assembly code, with some cooperation from the programmer.

Many algorithms in cryptography have clearly demarcated stages. One clear take-away point is that in order to verify such algorithms, enhanced compositional reasoning techniques that take full advantage of such structures is needed. We try to provide this requisite enhanced compositional reasoning with new `cuts` and Ghost variables functionality.

Future work. The six instances in this work are just the beginning. The same technique applies to also any implementation of small ideal-lattice-based cryptosystems that also has NTT-based arithmetic, e.g., the KEMs NTRU Prime, LAC, or NewHope [LLZ+18, PAA+20, BBC+20] and the signatures Dilithium and Falcon [ABD+20a, PFH+20]. There are also a myriad of other architectures and other parameter sets to consider.

We could also envision extending CryptoLine to other PQCs such as Rainbow/UOV and Classic McEliece. Ideally, We would hope that CryptoLine and similar tools would make it safe to deploy high-speed custom-made assembly for PQC in production scenarios.

Appendix B

Large Integer Multiplications

B.1 Paper: Efficient Multiplication of Somewhat Small Integers using Number-Theoretic Transforms

Conventional wisdom purports that FFT-based integer multiplication methods (such as the Schönhage–Strassen algorithm) begin to compete with Karatsuba and Toom–Cook only for integers of several tens of thousands of bits. In this work, we challenge this belief, leveraging recent advances in the implementation of number-theoretic transforms (NTT) stimulated by their use in post-quantum cryptography. We report on implementations of NTT-based integer arithmetic on two Arm Cortex-M CPUs on opposite ends of the performance spectrum: Cortex-M3 and Cortex-M55. Our results indicate that NTT-based multiplication is capable of outperforming the big-number arithmetic implementations of popular embedded cryptography libraries for integers as small as 2048 bits. To provide a realistic case study, we benchmark implementations of the RSA encryption and decryption operations.

B.1.1 Introduction

The development of fast algorithms for arithmetic on big numbers is a well-established field of research. As with any computational problem, its study can be dissected into two parts: First, the analysis of the *asymptotic* complexity. Second, the analysis of *concrete* complexity for a chosen size of input. The results are often different: An algorithm may have inferior asymptotic performance but superior practical performance for a certain input size. The analysis of the "crossover point," that is, the input size at which an asymptotically faster algorithm also becomes practically faster, is an important question when moving from theory to practice. The present paper is about the evaluation of such crossover points in the case of big number arithmetic on microcontrollers.

The multiplication of big numbers can be performed in a variety of ways of decreasing asymptotic complexity and (unsurprisingly) increasing sophistication. At the base, the so-called "schoolbook multiplication" approaches calculate the product of two n-limb numbers $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$ by computing and accumulating all n^2 subproducts $a_i b_j$. While from a practical perspective, a lot of research has been conducted on the optimal *concrete* strategy, they all lead to an asymptotic complexity of $\mathcal{O}(n^2)$. Next, the Karatsuba method [KO62] and its generalization by Toom–Cook [Too63] lower the asymptotic complexity to $\mathcal{O}(n^{1+s})$ for varying $0 < s < 1$; for example, Karatsuba's method of computing

$$(a_0 + ta_1)(b_0 + tb_1) = a_0 b_0 + t^2 a_1 b_1 + t((a_0 + a_1)(b_0 + b_1) - a_0 a_0 - a_1 b_1)$$

leads to an asymptotic complexity of $\mathcal{O}(n^{\log_2 3}) \subseteq \mathcal{O}(n^{1.585})$. Moving further, starting with the famous Schönhage–Strassen [SS71], FFT-based integer multiplications achieve asymptotic complexity $\mathcal{O}(n \log n \log \log n)$ and better [Für09], and the long conjectured (and presumably final) complexity of $\mathcal{O}(n \log n)$ was only recently achieved in [HvdH21].

Despite its far superior asymptotic complexity, NTT-based integer multiplication is not used for number ranges found in contemporary asymmetric cryptography: In fact, quadratic multiplication strategies appear to be the most prominent choice in those contexts. At the same time, the past years have seen significant research and progress regarding fast implementation of the NTT, stimulated by their prominence in post-quantum cryptography. The primary objective of this paper is to evaluate how those optimizations affect the practical performance and viability of NTT-based big number arithmetic.

B.1.2 Results

We find that the crossover point for viability of NTT-based modular arithmetic is at around 2048 bits. More precisely, we compare to modular arithmetic implementations found in the popular TLS libraries BearSSL and Mbed TLS, and find that our NTT-based implementation outperforms both by 1.3×–2.2× on Cortex-M3 and by 1.8×–6.4× on Cortex-M55. We also notice that there is considerable optimization potential for the schoolbook multiplications in

BearSSL and Mbed TLS — when this is implemented, 2048-bit NTT-based modular multiplication is only slightly better (1.1×) than schoolbook multiplication on Cortex-M3, and essentially equal on Cortex-M55. When moving to 4096-bit multiplication, however, our NTT-based implementation outperforms even those highly optimized schoolbook multiplications. We thus think that NTT-based modular arithmetic should be considered from 2048-bit onward.

Software: All our Cortex-M3 code is available at `https://github.com/ntt-int-mul/ntt-int-mul-m3`. Our Cortex-M55 integer-multiplication code is available at `https://gitlab.com/arm-research/security/pqmx`.

Related work. Present-day general-purpose computer algebra systems switch to FFT-based multiplication only for very large numbers. GMP [Fre] uses Schönhage–Strassen when multiplying numbers with more than 3000–10000 limbs (i.e., at least 96 000 bits) depending on the platform.[1] However, when tailoring an implementation to a specific integer size and platform, the crossover point appears to be lower. Previous work on implementing RSA using Schönhage–Strassen [GKZ07] in hardware concluded that it can only outperform Karatsuba and Toom–Cook for key sizes larger than 48 000 bits. [Gar07] reports similar findings: It estimates Schönhage–Strassen to be competitive only for RSA key sizes above $2^{17} \approx 131\,000$ bits, several orders of magnitude beyond typical RSA parameter choices. To the best of our knowledge, there is no competitive implementation of real-world RSA using FFT-based integer multiplication.

Other work. In addition to improvements to the efficiency of number-theoretic transforms, post-quantum cryptography has stimulated research into efficient schoolbook multiplication strategies for integers of a few hundred bits, as found in elliptic-curve or isogeny cryptography. It would be interesting to study and compare the performance of RSA based on the combination of Karatsuba and those new quadratic multiplication algorithms. Another avenue for further research is the evaluation of NTT-based arithmetic on high-end processors.

---

[1] `https://gmplib.org/manual/FFT-Multiplication`

B.1.3 Preliminaries

B.1.3.1 RSA

The RSA (Rivest–Shamir–Adleman) cryptosystem [RSA78] was the most common public-key cryptosystem for decades and remains in widespread use, primarily with keys of 2048, 3072, or 4096 bits. We briefly recap how it works.

During key generation, a semiprime $N = pq$ with p and q of roughly equal size is generated. The public key is N and a small e to which power it is easy to raise, commonly $e = 2^{16} + 1$. We have $x^{k\phi(N)+1} \equiv x \pmod{N}$ for all x, k, where $\phi(N) = (p-1)(q-1)$ is the Euler's totient function. With $d \equiv e^{-1} \pmod{\phi(N)}$, the public map $x \mapsto x^e \bmod N$ is then inverted by the secret map $y \mapsto y^d \bmod N$, the secret key being d. Both encryption and signing primitives can be constructed based on this pair of public/private maps.

The private map can be evaluated using the Chinese Remainder Theorem (CRT) method, computing $x = y^d \bmod N$ by interpolating $x \equiv y^{d \bmod (p-1)} \pmod{p}$ and $x \equiv y^{d \bmod (q-1)} \pmod{q}$. Modular multiplications are commonly implemented using Montgomery multiplication, and modular exponentiation uses windowing methods.

B.1.3.2 FFT-Based Integer Multiplication

Numerous versions of FFT-based integer multiplications are known, but their blueprint is typically the following: First, find an FFT-based quasi-linear time multiplication algorithm in a suitable polynomial ring. Second, find a means to reduce integer multiplication to the chosen kind of polynomial multiplications.

Starting with [SS71, Pol71], numerous instantiations of this idea have been developed, using polynomials over $\mathbb{C}$, finite fields $\mathbb{F}_q$, integers modulo Fermat numbers $\mathbb{Z}/(2^{2^n} + 1)\mathbb{Z}$, and also multivariate polynomial rings [HvdH21]. Here, we focus on NTT-based integer multiplication using polynomials in $\mathbb{Z}_q[X]/(X^n - 1)$ with q a prime or bi-prime, which is close to [Pol71]. While variable-size integer multiplication requires recursive application of the above principle, it is not necessary for the integer sizes considered here.

Section B.1.3.3 discusses how the NTT yields a quasi-linear multiplication in $\mathbb{Z}_q[X]/(X^n - 1)$. We now explain the reduction from integer multiplication.

To turn a multiplication of $a, b \in \mathbb{Z}$ into a multiplication in $\mathbb{Z}_q[X]/(X^n - 1)$, one first lifts a, b to integer *polynomials* $A, B \in \mathbb{Z}[X]$ along $f : \mathbb{Z}[X] \to \mathbb{Z}, X \mapsto 2^\ell$, the canonical choice being the radix-2^ℓ presentations of a, b. Since $f(AB) = ab$, it suffices to compute $AB \in \mathbb{Z}[X]$. To do so, one chooses q and n such that $AB \in \mathbb{Z}[X]$ is a canonical representative for the finite quotient $\mathbb{Z}_q[X]/(X^n - 1)$, that is, it is of degree $< n$ with coefficients in $\{0, \ldots, q-1\}$. Under these circumstances, one can then uniquely recover AB from its image $g(AB) = g(A)g(B)$ under $g : \mathbb{Z}[X] \to \mathbb{Z}_q[X]/(X^n - 1)$. We have thus reduced the computation of ab in $\mathbb{Z}$ to that of $g(A)g(B)$ in $\mathbb{Z}_q[X]/(X^n - 1)$.

B.1.3.3 Number-Theoretic Transforms

The *number-theoretic transform* (NTT) is a generalization of the discrete Fourier transform, replacing the base ring $\mathbb{C}$ of the complex numbers by other commutative rings, commonly finite fields $\mathbb{F}_q$. In the present context, its value lies in the fact that it transforms convolutions into point-wise products in quasi-linear time, reducing the complexity of convolutions from quadratic to quasi-linear.

Definition. We're working over $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ for odd q and fix $\omega \in \mathbb{Z}_q$ an nth root of unity. We write $[n] = \{0, 1, \dots, n-1\}$. The NTT [Für09, HvdH21] is the canonical projection $\mathbb{Z}_q[x]/\langle x^n - 1\rangle \to \prod_i \mathbb{Z}_q[x]/\left\langle x - \omega^i \right\rangle$, which under the isomorphism $\mathbb{Z}_q[x]/\left\langle x - \omega^i \right\rangle \cong \mathbb{Z}_q, \boldsymbol{a}(x) \mapsto \boldsymbol{a}(\omega^i)$ can also be described as

$$\mathsf{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \to \mathbb{Z}_q^n, \quad \mathsf{NTT}(a) = \left(\boldsymbol{a}(1), \boldsymbol{a}(\omega), \dots, \boldsymbol{a}(\omega^{n-1})\right).$$

If ω is a principal nth root of unity and n is invertible in $\mathbb{Z}_q$, this constitutes a ring isomorphism $\mathsf{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$; in particular, we have $ab = \mathsf{NTT}^{-1}\left(\mathsf{NTT}(a) \cdot_{\Pi} \mathsf{NTT}(b)\right)$, where $\cdot_{\Pi}$ is the point-wise multiplication in $\mathbb{Z}_q^n$.

Fourier inversion. Domain and codomain of the NTT can be identified via the isomorphism of $\mathbb{Z}_q$-modules (not rings) $\varphi : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$, $x^i \leftrightarrow e_i$ (where e_i is the ith unit vector). This renders the resulting $\mathsf{NTT} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ close to an involution: $\mathsf{NTT}^2 = \mathrm{mul}_n \circ \mathtt{neg}$, where $\mathrm{mul}_n : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ is point-wise multiplication with n and $\mathtt{neg} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ sends e_i to $e_{\mathtt{neg}(i)}$ with $\mathtt{neg}(0) = 0$ and $\mathtt{neg}(i) = n - i$ for $i > 0$ (we do not distinguish between a permutation on $[n]$ and the induced isomorphism on $\mathbb{Z}_q^n$). Another way of saying this is that $\mathsf{NTT}' : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ defined by $\mathsf{NTT}'(a) = \Big(\boldsymbol{a}(1), \boldsymbol{a}(\omega^{-1}), \dots, \boldsymbol{a}(\omega^{-(n-1)})\Big)$ is, up to multiplication by n and application of φ, the inverse of $\mathsf{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$. This is the *Fourier Inversion Formula*, and the curious reader will find that it boils down to the orthogonality relations $\sum_j \omega^{ij} = n \cdot \delta_{i,0}$.

Fast Fourier transform. The NTT can be calculated using the Cooley–Tukey (CT) FFT algorithm: For $n = 2m$, CT splits $\mathbb{Z}_q[x]/\left\langle x^{2m} - \zeta^2 \right\rangle$ into $\mathbb{Z}_q[x]/\langle x^m - \zeta\rangle \times \mathbb{Z}_q[x]/\langle x^m + \zeta\rangle$ via $\mathrm{CT}(a + x^m b, \zeta) = (a + \zeta b, a - \zeta b)$ for a, b of degree $< m$ — this is called a *CT butterfly*. The idea can be applied recursively, and for $n = 2^k$ we obtain a map $\mathsf{NTT}_{\mathsf{CT}} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ which is equal to $\texttt{bitrev} \circ \mathsf{NTT}$, where $\texttt{bitrev} : [2^k] \to [2^k]$ is the bit-reversal permutation.

The CT strategy can also be applied for radices $r \neq 2$, performing one splitting $\mathbb{Z}_q[x]/\langle x^{rm} - \zeta^r\rangle \cong \prod_i \mathbb{Z}_q[x]/\langle x^m - \omega_r^i \zeta\rangle$ at a time. When applied recursively to a factorization $n = r_1 \cdots r_s$, the resulting map $\mathsf{NTT}_{\mathsf{CT}} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ agrees with $\sigma(r_1, \ldots, r_s) \circ \mathsf{NTT}$, where $\sigma(r_1, \ldots, r_s)$ is given by

$$[n] \quad \cong \quad [r_1] \times \ldots \times [r_s] \quad \xrightarrow{\text{reverse}} \quad [r_s] \times \ldots \times [r_1] \quad \cong \quad [n]$$

where the first and last map are lexicographic orderings. Note that $\sigma(2, \ldots, 2) = \texttt{bitrev}$, and $\sigma(r_1, \ldots, r_s)$ is an involution only if $(r_1, \ldots, r_s)$ is a palindrome.

Inverse NTT. There are two approaches for implementing $\mathsf{NTT}_{\mathsf{CT}}^{-1}$: First, one can invert CT butterflies via *Gentleman–Sande butterflies* $\mathrm{GS}(a, b, \zeta) = (a + b, (a - b)\zeta)$. Alternatively, one can leverage $\mathsf{NTT}_{\mathsf{CT}} = \sigma \circ \mathsf{NTT}$ and $\mathsf{NTT}^{-1} = \mathrm{mul}_{1/n} \circ \mathsf{NTT}'$ to compute $\mathsf{NTT}_{\mathsf{CT}}^{-1} = \mathrm{mul}_{1/n} \circ \mathsf{NTT}' \circ \sigma^{-1} = \mathrm{mul}_{1/n} \circ \sigma^{-1} \circ \mathsf{NTT}'_{\mathsf{CT}} \circ \sigma^{-1}$. If σ is an involution (e.g., if $n = 2^k$), this is $\mathrm{mul}_{1/n} \circ \sigma \circ \mathsf{NTT}'_{\mathsf{CT}} \circ \sigma^{-1}$ and can thus be implemented like $\mathsf{NTT}_{\mathsf{CT}}$ while implicitly applying the permutation σ; this leads to the implementation of $\mathsf{NTT}_{\mathsf{CT}}^{-1}$ as presented in [ACC+21, Figure 1], which does not require explicit permutations. For a general mixed-radix NTT, however, σ is not an involution, and an explicit permutation by σ^{-2} is needed; we avoid this via Good's trick, as explained in the next section.

GS butterflies lead to exponential growth for an exponentially shrinking number of coefficients, while CT butterflies yield linear growth for *all* coefficients. This impacts the amount and placement of reductions during $\mathsf{NTT}^{\pm 1}$.

Good's trick. For $n = rs$ with coprime r, s, another strategy to computing NTT_n is computing the bottom edge in the commutative diagram

$$\begin{tikzcd}
\frac{\mathbb{Z}_q[x]}{\langle x^n - 1\rangle} \arrow[rr, "\mathsf{NTT}_n^{\omega}"'] \arrow[d, "k\equiv i \pmod r \atop k\equiv j \pmod s"', "u^i \otimes v^j \leftrightarrow x^k \atop \cong"] & & \mathbb{Z}_q^n \arrow[d, dash, "e_i \otimes e_j \leftrightarrow e_k \atop \cong"', "k\equiv i \pmod r \atop k\equiv j \pmod s"] \\
\frac{\mathbb{Z}_q[u]}{\langle u^r - 1\rangle} \otimes_{\mathbb{Z}_q} \frac{\mathbb{Z}_q[v]}{\langle v^s - 1\rangle} \arrow[rr, "\mathsf{NTT}_r^{\omega^e} \otimes \mathsf{NTT}_s^{\omega^f}", "e\equiv 0 \pmod s,\ e\equiv 1 \pmod r \atop f\equiv 1 \pmod s,\ f\equiv 0 \pmod r"'] & & \mathbb{Z}_q^r \otimes_{\mathbb{Z}_q} \mathbb{Z}_q^s
\end{tikzcd}$$

There are two benefits: First, if r, s are prime powers then $\mathsf{NTT}_{r/s}^{\pm 1}$ can be computed via CT as described above with the permutation an involution. Second, fewer twiddle factors are needed for the computation of $\mathsf{NTT}_s \otimes \mathsf{NTT}_r$.

Incomplete NTTs. Denoting $R = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ and $R_i = \mathbb{Z}_q[x]/\big\langle x - \omega^i\big\rangle$. The NTT splitting $\texttt{NTT} : R \xrightarrow{\cong} \prod_i R_i$ transfers to any R-algebra: If S is an R-algebra, we have $S \cong S \otimes_R R \cong S \otimes_R \prod_i R_i \cong \prod_i S \otimes_R R_i$. The most common example are *incomplete NTTs*: The ring $S = \mathbb{Z}_q[y]/\big\langle y^{nh} - 1\big\rangle$ is an algebra over its subring $R = \mathbb{Z}_q[y^h]/\big\langle y^{nh} - 1\big\rangle \cong \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ to which the NTT applies, and so $S \cong \prod_i S \otimes_R \mathbb{Z}_q[y^h]/\big\langle y^h - \omega^i\big\rangle = \prod_i \mathbb{Z}_q[y]/\big\langle y^h - \omega^i\big\rangle$.

The benefits of using incomplete NTTs are: First, we only need an nth principal root of unity to partially split $\mathbb{Z}_q[y]/\big\langle y^{nh} - 1\big\rangle$. Second, polynomial multiplication using incomplete NTTs and "base multiplication" in $\mathbb{Z}_q[y]/\big\langle y^h - \omega^i\big\rangle$ may be faster than for full NTTs and base multiplication in $\mathbb{Z}_q$. We use incomplete NTTs for all parameter sets — see below.

Fermat number transforms. The Fermat number transform (FNT) is a special case of NTT where the modulus is a Fermat number $F_t = 2^{2^t} + 1$ [AB74]. For the coefficient ring $\mathbb{Z}_{F_t}$, we can compute a size-n NTT if n divides 2^{t+2}. If we choose 2 to be the principal 2^{t+1}th root of unity, then the twiddle factors for a size-(t+1) Cooley–Tukey FFT are all powers of 2.

Since there are square roots for ±2, we can choose a principal 2^{t+2}th root of unity ω with $\omega = \sqrt{2}$ and compute a size-2^{t+2} NTT [AB74]. Furthermore, if F_t is a prime, then we can compute a size-2^{2^t} NTT. Note that the only known prime Fermat numbers are $F_0, \ldots, F_4$.

B.1.3.4 Modular Reductions and Multiplications

(Refined) Barrett reduction. Signed Barrett reduction approximates

$$a \bmod^{\pm} q = a - q\,\lfloor a/q \rceil = a - q\,\left\lfloor a\tfrac{\mathtt{R}}{q}/\mathtt{R} \right\rceil \approx a - q \left\lfloor \frac{a\left[\!\left[\frac{\mathtt{R}}{q}\right]\!\right]}{\mathtt{R}} \right\rceil$$

where $\mathtt{R} = 2^w$ is a power of 2 and $[\![\mathtt{R}/q]\!]$ is a precomputed integer approximation to $\frac{\mathtt{R}}{q}$. The quality of the resulting approximation $\left\lfloor \frac{a\left\lfloor\frac{\mathtt{R}}{q}\right\rceil}{\mathtt{R}} \right\rceil \approx a \bmod^{\pm} q$ — and in particular, the question of when it may in fact be an *equality* — depends on the value of w, and two choices for w are common, as we now recall.

First, $w = M$ where $M \in \{16, 32\}$ is the word or half-word size, allowing $\left\lfloor \frac{-}{\mathtt{R}} \right\rceil$ to be conveniently implemented using rounding high multiply instructions. We call this the "standard" Barrett reduction.

Second, $w = (M - 1) + \lfloor \log_2 q \rfloor$, which is maximal under the constraint that $[\![\mathtt{R}/q]\!]$ is a signed M-bit integer: This choice leads to higher accuracy of the approximation, but typically results in an additional instruction. We will henceforth call it the "refined" Barrett reduction. For standard Barrett reduction, both $\llbracket a \rrbracket = 2\left\lfloor \frac{a}{2} \right\rceil$ and $\llbracket a \rrbracket = \lfloor a \rceil$ can be useful, while for refined Barrett reduction, we always choose $\llbracket a \rrbracket = \lfloor a \rceil$ because of its tighter bound $|\lfloor a \rceil - a| \leq \frac{1}{2}$.

Note that both "standard" and "refined" Barrett reductions are already known in the literature as Barrett reductions. We make this distinction for introducing an extension of the signed Barrett multiplication by [BHK$^+$21].

(Refined) Barrett multiplication. For two integers a, b and a modulus q, signed Barrett multiplication [BHK$^+$21] approximates

$$ab \bmod^{\pm} q = ab - q\left\lfloor \frac{ab}{q} \right\rceil = ab - q\left\lfloor a\frac{b\mathtt{R}}{q}/q \right\rceil \approx ab - \left\lfloor \frac{a \cdot \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket}{\mathtt{R}} \right\rceil q,$$

where again $\mathtt{R} = 2^w$ is a power of 2 and $\llbracket b\mathtt{R}/q \rrbracket$ is a precomputed integer approximation to $\frac{b\mathtt{R}}{q}$. Previously, only the choice $w = M \in \{16, 32\}$ was considered. In analogy with refined Barrett reduction, we suggest to also consider $w = (M-1) + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$, which again is maximal under the constraint that $\llbracket b\mathtt{R}/q \rrbracket$ is a signed M-bit integer. We call the resulting approximation to $ab \bmod^{\pm} q$ the "refined" Barrett multiplication.

We summarize the quality and size of Barrett reduction and multiplication:

Lemma 6. Let $q \in \mathbb{N}$ be odd and $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^{M-1}$ for $M \in \{16, 32\}$. Moreover, let $\llbracket - \rrbracket : \mathbb{Q} \to \mathbb{Z}$ be any integer approximation, i.e. $|x - \llbracket x \rrbracket| \leq 1$ for all $x \in \mathbb{Q}$, and put $t \bmod^{\llbracket\ \rrbracket} q = t - q\llbracket t/q \rrbracket$. Then for $\mathtt{R} = 2^M$ we have

$$|ab - \left\lfloor \frac{a \cdot \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket}{\mathtt{R}} \right\rceil q| \leq \frac{a(b\mathtt{R} \bmod^{\llbracket\ \rrbracket} q)}{\mathtt{R}} + \frac{\mathtt{R}}{2}.$$

Proof. See [BHK$^+$21, Corollary 2] □

Lemma 7. Let $q \in \mathbb{N}$ be odd and $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^{M-1}$ for $M \in \{16, 32\}$. Moreover, pick $k \geq 1$ maximal s.t. $\varepsilon := |\lfloor b\mathtt{R}/q \rceil - b\mathtt{R}/q| \leq 2^{-k}$. Finally, set $\mathtt{R} = 2^w$ for $w = (M-1) + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$. If $\log_2 |a| < (M-1) - (\lceil \log_2 |b| \rceil - (k-1))$, then

$$ab - \left\lfloor \frac{a \cdot \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket}{\mathtt{R}} \right\rceil q = ab \bmod^{\pm} q.$$

Restating Lemma 7 in simple terms: Refined Barrett *reduction* (the special case $b = 1$) yields canonical representatives for *all* inputs a with $|a| < 2^{M-1}$. For a refined Barrett multiplication, the range of inputs for which $ab - \left\lfloor \frac{a\cdot \left\llbracket \frac{b\mathtt{R}}{q} \right\rrbracket}{\mathtt{R}} \right\rceil q$ is guaranteed to be canonical is narrowed by the bit-size of b; *however*, this can be compensated for by an exceptionally close approximation $b\mathtt{R}/q \approx \lfloor b\mathtt{R}/q \rceil$.

Proof of Lemma 7. Setting $\delta = a \lfloor b\mathtt{R}/q \rceil / \mathtt{R} - ab/q$, it follows from the definition of ε and k that $|\delta| \leq |a|/2^{k+w}$. Since $\lfloor - \rceil$ changes its value only when crossing values of the form $\{\frac{2n+1}{2}\}$ for $n \in \mathbb{Z}$, for $\left\lfloor \frac{ab}{q} \right\rceil$ and $\left\lfloor \frac{a\left\lfloor \frac{b\mathtt{R}}{q} \right\rceil}{\mathtt{R}} \right\rceil = \left\lfloor \frac{ab}{q} + \delta \right\rceil$ to agree it is sufficient to show that $|\delta| < \min\left\{ \left| \frac{2n+1}{2} - \frac{c}{q} \right| \mid c, n \in \mathbb{Z} \right\} = \frac{1}{2q}$ — the last equality holds since q is odd. Refined Barrett multiplication is thus guaranteed to yield the canonical representative of ab if $\frac{|a|}{2^{k+w}} < \frac{1}{2q}$, i.e. $|a| < \frac{2^{k+w-1}}{q}$. Plugging in $w = M - 1 + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$ and estimating $q < 2^{\lfloor \log_2 |q| \rfloor + 1}$, this follows provided $\log_2 |a| < (M-1) - (\lceil \log_2 |b| \rceil - (k-1))$, as claimed. ☐

Example 11. Let $M = 32$, $q = 114826273$, and $b = 774$. Then $\lfloor \log_2 q \rfloor = 26$ and $\lceil \log_2 b \rceil = 10$, so $w = 47$. Moreover, $\varepsilon = ||b\mathtt{R}/q\rceil - b\mathtt{R}/q|$ satisfies $\varepsilon < 2^{-11}$. Thus, according to Lemma 7, the refined Barrett multiplication with $\mathtt{R} = 2^{47}$ does therefore yield canonical representatives for all inputs a with $|a| < 2^{31}$: The exceptionally good approximation $\lfloor b\mathtt{R}/q \rceil \approx b\mathtt{R}/q$ makes up for the size of b.

Montgomery multiplication. The Montgomery multiplication [Mon85] of a, b with respect to a modulus q and a 2-power $\mathtt{R} > q$ is defined as

$$\mathtt{hi}\left(a \cdot b + q \cdot \mathtt{lo}\left(q' \cdot \mathtt{lo}\left(a \cdot b\right)\right)\right),$$

providing a representative of $ab\mathtt{R}^{-1}$ modulo q. Here, $q' = -q^{-1} \bmod \mathtt{R}$, and `lo` and `hi` are extractions of the lower and upper $\log_2 \mathtt{R}$ bits, respectively. Montgomery multiplication is defined and relevant for both small-width modular arithmetic such as modular arithmetic modulo a 16-bit or 32-bit prime, as well as large integer arithmetic as used, e.g., in RSA.

Multi-precision Montgomery multiplication. Montgomery multiplication for big integers is implemented iteratively: For $a, b = \sum_i b_i \mathtt{B}^i$, one computes a representative of $ab\mathtt{B}^{-n}$ by writing

$$ab\mathtt{B}^{-n} = \ldots(ab_2 + (ab_1 + (ab_0)\mathtt{B}^{-1})\mathtt{B}^{-1})\mathtt{B}^{-1}\ldots$$

and computing each $x \mapsto (x + ab_i)\mathtt{B}^{-1}$ using a Montgomery multiplication w.r.t. $\mathtt{B}$. Each such step involves the computation and accumulation of $P = x + ab_i$ and of $Q = ((x + ab_i)_0 q' \bmod \mathtt{B})p$. If the products are computed separately, this is called *Coarsely Integrated Operand Scanning* (CIOS) [KAK96]. If

$(x + ab_i)_0 q' \bmod \mathtt{B}$ is computed first and then $P + Q$ is computed in one loop, it is called *Finely Integrated Operand Scanning* (FIOS).

Divided-difference for Chinese remainder theorem (CRT). We compute polynomial products modulo $q_1 q_2$ by interpolating products modulo q_1 and q_2 using the divided-difference algorithm for CRT [CHK+21]: Let q_0, q_1 be two coprime integers and $m_1 := q_0^{-1} \bmod {}^{\pm}q_1$. For a system $u \equiv u_0 \pmod{q_0}, u \equiv u_1 \pmod{q_1}$ with $|u_0| < \frac{q_0}{2}, |u_1| < \frac{q_1}{2}$, we solve for u with $|u| < \frac{q_0 q_1}{2}$ by computing:

$$u = u_0 + \left((u_1 - u_0)m_1 \bmod {}^{\pm}q_1\right) q_0. \tag{B.1}$$

B.1.3.5 Implementation Targets

We briefly explain our choice of implementation targets.

B.1.3.6 Cortex-M3

The Arm® Cortex®-M3 CPU is a low-cost processor found in a wide range of applications such as microcontrollers, automotive body systems, or wireless networking. It implements the Armv7-M architecture and features a 3-stage pipeline, an optional memory protection unit (MPU) and a single-cycle $32 \times 32 \rightarrow 32$-bit multiplier with optional 1-cycle accumulation or subtraction.

We select the Cortex-M3 primarily for two reasons: First, it is a popular choice of MCU for automotive hardware security modules (e.g. Infineon AURIX TC27X). Second, its $32 \times 32 \rightarrow 64$ long multiplication instructions `smull`, `smlal`, `umull`, `umlal` have data-dependent timing and lead to timing side channels when used to process sensitive data. To avoid those, implementations need to use single-width multiplication instructions `mul`, `mla`, and `mls` instead. We expect this reduction of basic multiplication width to have a more significant impact on the runtime of classical multiplication than on (quasi-linear) NTT-based multiplication. A goal of the paper is to evaluate this intuitive assessment.

B.1.3.7 Cortex-M55

The Cortex-M55 processor is the first implementation of the Armv8.1-M architecture, with optional support for the M-Profile Vector Extension (MVE), or Arm® Helium™ Technology. It features a 5-stage pipeline when Helium is enabled, and except for some pairs of Thumb instructions, it is single issue. In addition to the Helium vector extension, it supports the Low Overhead Branch

Extension, as well as tightly coupled memory (TCM) for both code and data, with a total Data-TCM bandwidth of 128-bit/cycle, 64-bit/cycle for CPU processing and 64-bit/cycle for concurrent DMA transfers. For a more extensive introductions to both the Armv8.1-M architecture and the Cortex-M55 CPU, we refer to [BMK$^+$21, Section 3] and the references therein.

We select the Cortex-M55 for the following reasons: First, due to its support for SIMD vector processing, it is an exciting and powerful new implementation target — the cryptographic capabilities of which are still to be explored. Second, the authors are not aware of means to vectorize classical `umaal`-based multiplication strategies using MVE, while in contrast it has been demonstrated in [BMK$^+$21] that the NTT is amenable for significant speedup using MVE. We are thus curious to understand how a vectorized NTT-based integer multiplication fares compared to classical `umaal`-based integer multiplication.

B.1.4 Implementations

B.1.4.1 High-Level Strategy

We implement Montgomery multiplication on top of NTT-based large integer multiplication, the latter as described in Section B.1.3.2. This is in contrast to CIOS/FIOS approaches for iterative Montgomery multiplication, which never need to compute the double-width product of two large integers.

We pick $\texttt{R} = 2^{\ell \cdot n/2}$, which in contrast to $\texttt{R} = 2^N$ aligns taking the low and high half w.r.t. $\texttt{R}$ with taking the low resp. high halves of polynomials.

NTT-based large integer multiplication involves a considerable amount of precomputation, such as chunking and NTT. Since each Montgomery multiplication involves three integer multiplications — $a \cdot b$, $t = q' \cdot (a \cdot b)_{\text{low}}$, and $p \cdot t$ — two of which involve static factors p and p', we buffer their precomputations. We also make use of asymmetric multiplication [BHK$^+$21] and refer to the resulting NTT and base multiplication as $\mathsf{NTT}_{\text{heavy}}$ and $\texttt{basemul}_{\text{light}}$.

Algorithm B.1, Algorithm B.2 and Appendix B.1.6.3 describe our modular multiplication strategy in more detail. Appendix B.1.6.2 explains how to perform the non-trivial precomputation of $p^{-1} \bmod \texttt{R}$ for our large choice of $\texttt{R}$.

Algorithm B.1 Montgomery squaring using NTTs.

```
Input: $p, a\mathtt{R} \bmod p, \hat{p}^{-1} = \mathsf{NTT}(\mathtt{chk}(p^{-1} \bmod \mathtt{R})), \hat{p} = \mathsf{NTT}(\mathtt{chk}(p))$.
Output: $c = a^2\mathtt{R} \bmod p$.
$\hat{a} = \mathsf{NTT}(\mathtt{chk}(a))$
$t = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{a}\hat{a}))$
$\hat{t} = \mathsf{NTT}(\mathtt{chk}(t \bmod \mathtt{R}))$
$l = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{t}\hat{p}^{-1}))$
$\hat{l} = \mathsf{NTT}(\mathtt{chk}(l \bmod \mathtt{R}))$
$r = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{l}\hat{p}))$
$c = \frac{t}{\mathtt{R}} - \frac{r}{\mathtt{R}}$
if $c < 0$ then $c = c + p$
return $c$
```

Algorithm B.2 Montgomery multiplication using NTTs.

```
Input: $a\mathtt{R} \bmod p, b\mathtt{R} \bmod p, \hat{p}^{-1} = \mathsf{NTT}(\mathtt{chk}(p^{-1} \bmod \mathtt{R})), \hat{p} = \mathsf{NTT}(\mathtt{chk}(p))$.
Output: $c = ab\mathtt{R} \bmod p$.
$\hat{a} = \mathsf{NTT}(\mathtt{chk}(a))$
$\hat{b} = \mathsf{NTT}(\mathtt{chk}(b))$
$t = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{a}\hat{b}))$
$\hat{t} = \mathsf{NTT}(\mathtt{chk}(t \bmod \mathtt{R}))$
$l = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{t}\hat{p}^{-1}))$
$\hat{l} = \mathsf{NTT}(\mathtt{chk}(l \bmod \mathtt{R}))$
$r = \mathtt{dechk}(\mathsf{NTT}^{-1}(\hat{l}\hat{p}))$
$c = \frac{t}{\mathtt{R}} - \frac{r}{\mathtt{R}}$
if $c < 0$ then $c = c + p$
return $c$
```

B.1.4.2 Parameter Choices

Recall from Section B.1.3.2 that the Schönhage–Strassen algorithm involves lifting N-bit numbers to $\mathbb{Z}[X]$ along $X \mapsto 2^\ell$ and computing their product in $\mathbb{Z}_q[X]/(X^n - 1)$ using the NTT. We now describe our choices of N, ℓ, n, q; they were found by manually tailoring the algorithm to the given target architectures.

Firstly, if we divide our inputs into ℓ-bit chunks, we need $n \geq 2\left\lceil \frac{N}{\ell} \right\rceil$; otherwise, we cannot lift from $\mathbb{Z}_q[X]/(X^n - 1)$ back to $\mathbb{Z}_q[X]$. For performance, we also want n to be a multiple of a power of two so that NTT-based polynomial multiplication is fast. Hence, we may deliberately choose $n > 2\left\lceil \frac{N}{\ell} \right\rceil$ and pad with zeros when needed.

Secondly, the coefficients of the product of two dimension-$(n/2)$ polynomials with ℓ-bit coefficients are bounded by $\frac{n}{2} \cdot 2^{2\ell}$, so we need $q \geq \frac{n}{2} \cdot 2^{2\ell}$ to be able to lift from $\mathbb{Z}_q[X]$ back to $\mathbb{Z}[X]$. However, we also need to pick q so that $\mathbb{Z}_q$ has a principal nth root of unity, as otherwise the NTT is not defined. We pick $q = q_1 q_2$ a bi-prime and compute modulo q_1 and q_2 separately via CRT; it is more preferable to map two half-size moduli maps to the available hardware multipliers instead off mapping a single larger q. Table B.1 presents our choices, and we explain them in detail now.

On the Cortex-M3, we use chunks of $\ell = 11$ bits, so $\left\lceil \frac{N}{\ell} \right\rceil = 187$ for $N = 2048$ and $\left\lceil \frac{N}{\ell} \right\rceil = 373$ for $N = 4096$, but pick slightly larger $n = 384 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 2048$ and $n = 768 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 4096$ since both are dimensions for which a fast NTT can be implemented. Next, we need $q_1 q_2 \geq 192 \cdot 2^{22}$ for $N = 2048$ and $q_1 \cdot q_2 \geq 384 \cdot 2^{22}$ for $N = 4096$; we pick $(q_1, q_2) = (12289, 65537)$ for $N = 2048$, and $(q_1, q_2) = (25601, 65537)$ for $N = 4096$. The Fermat prime $q_2 = 65537$ allows particularly fast NTT computation using the FNT, while the other prime is chosen to be the smallest admissible prime for which a 128th (resp. 256th) principal root of unity exists.

On the Cortex-M55, we use chunks of $\ell = 22$ bits, so $\left\lceil \frac{N}{\ell} \right\rceil = 94$ for $N = 2048$ and $\left\lceil \frac{N}{\ell} \right\rceil = 187$ for $N = 4096$, but again pick slightly larger $n = 192 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 2048$ and $n = 384 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 4096$ since those are NTT-friendly dimensions. For $q = q_1 q_2$, we pick $114\,826\,273 \cdot 128\,919\,937$ for both $N = 2048$ and $N = 4096$. Those choices are motivated as follows: First, we have $q \approx 2^{53.7} > 2^{51.58} \approx \frac{384}{2} \cdot 2^{44}$. In fact, since we even have $q > 4 \cdot (\frac{384}{2} \cdot 2^{44})$, we can recover the coefficients in the *sum* of two polynomial products as the *signed* canonical representatives of their image in $\mathbb{Z}_q$. The former allows saving one CRT during the Montgomery multiplication, while the latter means that we do not need a signed-to-unsigned conversion after the signed CRT. Second, q_1, q_2 are carefully chosen so that $(q_2 \bmod q_1)^{-1}$ in $\mathbb{Z}_{q_2}$ is amenable to refined Barrett multiplication — in fact, since $(q_2 \bmod q_1)^{-1} = 774$, this is what we observed in Example 11. Thirdly, both $q_1 - 1$ and $q_2 - 1$ are multiples of 96 and thus support incomplete dimension-96 NTTs. Finally, $q_1, q_2 < 2^{27}$ are small enough that during the dimension-96 NTTs, no explicit modular reduction is required.

Table B.1: Parameters of NTT multiplications for RSA. N stands for the target bit-size, ℓ stands for the chunk size, and n stands for the dimension of the polynomial ring.

N	ℓ	n	NTT	Modulus $q = q_1 \cdot q_2$
Cortex-M3				
2048	11 bits	384	$128 = 2^7$	$12289 \cdot 65537$
4096	11 bits	768	$256 = 2^8$	$25601 \cdot 65537$
Cortex-M55				
2048	22 bits	192	$64 \cdot 3 = 2^6 \cdot 3$	$114\,826\,273 \cdot 128\,919\,937$
4096	22 bits	384	$128 \cdot 3 = 2^7 \cdot 3$	$114\,826\,273 \cdot 128\,919\,937$

B.1.4.3 Chunking and Dechunking

We need to convert between multi-precision integers and polynomials, which we refer to as “chunking” `chk`() and “dechunking” `dechk`(). `chk`() takes an N-bit multi-precision integer and splits it into n chunks of ℓ bits each, viewed as the coefficients of a polynomial. In other words, we lift along $\mathbb{Z}[X] \to \mathbb{Z}, X \mapsto 2^\ell$. `dechk`() converts a polynomial to a multi-precision integer by evaluating the polynomial at $X = 2^\ell$. As the coefficients of polynomials may grow beyond 2^ℓ during computation, this requires carrying through the entire polynomial and packing into a multi-precision integer.

B.1.4.4 Modular Exponentiation and Table Lookup

For the private-key operations, we use square-and-multiply with Algorithms B.1 and B.2 to implement constant-time exponentiation with a fixed window size of w bits. This requires constant-time table lookups, and choosing the optimal w depends on the relative costs of a modular multiplication compared to such lookups: The cost of a lookup scales linearly in the table size 2^w, whereas the number of required multiplications only scales proportionally to $1/w$. We have determined that $w = 6$ is the fastest choice for both Cortex-M3 and Cortex-M55 and both 2048 and 4096 bits. We note that memory consumption will be an increasing concern as w grows, since the lookup table contains 2^w entries — exponentially large in w. In turn, reducing w will incur only a mild performance penalty while allowing for a significant reduction in the table size.

It may seem at first that storing the table entries in NTT domain should be preferable. However, the much larger size of elements in NTT domain results in drastically slower table lookups, which in our implementation clearly outweighs the cost of transforming to NTT domain on the fly after each load. Thus, our implementation stores the table entries as integer values.

For the public-key operation, we use a straightforward square-and-multiply with the fixed public exponent $2^{16} + 1$ which is overwhelmingly common in practice.

B.1.4.5 Implementation Details for Cortex-M3

Our Cortex-M3 NTT implementation relies on a code generator written in Python, featuring a bound-checker determining when it should insert reductions, and which aborts if it cannot guarantee the correctness of the computation. The result is a set of fully unrolled assembly implementations of NTT, inverse NTT, base multiplication and squaring, for configurable moduli.

The code generator uses the same high-level structure for FNTs and "generic" NTTs, the main difference being in the reductions. The generator also recognizes multiplications by power-of-two constants and converts them to shifts when appropriate; this is one of the main optimizations employed by FNTs.

Number-theoretic transforms. We use incomplete NTTs of lengths $384 = 2^7 \cdot 3$ and $768 = 2^8 \cdot 3$. Both NTTs are implemented for the prime moduli $q_1 = 12289$ ($q_1 = 25601$) and $q_2 = 65537$, which by CRT correspond to a single NTT of the same length modulo $q_1 q_2$. We use CT butterflies for the forward NTT and GS butterflies for the inverse. Layers are merged as appropriate[2] to eliminate unnecessary store-load pairs. The base multiplication is a straightforward polynomial multiplication in a ring of the form $\mathbb{Z}_q[X]/(X^3 - \zeta)$. The CRT computation after the inverse NTTs is applied to each coefficient separately and follows Equation B.1.

"General" number-theoretic transform. For most moduli such as $q_1 = 12289$ and $q_1 = 25601$, we use a combination of (signed) Montgomery multiplication (Appendix B.1.6.1, Algorithm B.4) and (signed) Barrett reductions (Appendix B.1.6.1, Algorithm B.3). The Barrett reduction comes in two variants, the difference consisting in the optional addition of $\mathtt{R}/2$ before the right shift. Skipping the addition is faster, but results in worse reduction quality.

---

[2] The layers are merged as $4+3$ resp. $4+2+2$ in the forward NTTs, exploiting that the upper half of the input coefficients are zero, and $3+2+2$ resp. $3+3+2$ in the inverse NTTs. Register pressure prohibits more aggressive merging.

Fermat number transform. For the Fermat prime $q_2 = 2^{16}+1 = 65537$, we use variants of the "FNT reduction" shown in Appendix B.1.6.1, Algorithm B.5. Depending on the desired reduction quality, the algorithm is either applied (1) as written, or (2) with its input offset by 2^{15} and the output correspondingly offset by -2^{15}, or (3) followed by a conditional subtraction of $2^{16}+1$ if the output is $> 2^{15}$. Method (2) produces a representative in $\{-2^{16}+1, 2^{16}-1\}$, while the output of method (3) is a canonical symmetric representative, i.e., lies in $\{-2^{15}, ..., 2^{15}\}$. Methods (2) and (3) are equally fast if the constant 2^{15} can be kept in a low register throughout. If register pressure renders this undesirable, method (2) provides a convenient "intermediate" solution between the very fast FNT reduction and the canonical symmetric reduction.

Constant-time lookup. We use predicated moves to extract the desired table entry in a "striding" fashion: For each slice of four 32-bit words, the respective part of each table entry is loaded and conditionally moved into a set of target registers using a `itttt eq; moveq; moveq; moveq; moveq` instruction sequence. The target registers are stored after processing all entries. Compared to the alternative of traversing the table entry by entry, this finalizes each output word immediately, and no partial outputs have to be stored and reloaded.

B.1.4.6 Implementation Details for Cortex-M55

Pipeline efficiency. As explained in [BMK+21], Cortex-M55 is a *dual-beat* implementation of MVE; that is, most MVE instructions execute over two cycles. To still achieve a Instructions per Cycle (IPC) rate of more than 0.5 without costly dual-issuing logic, Cortex-M55 supports *instruction overlapping* for neighboring vector instructions, provided they use different execution resources. The balance and ordering of instructions is therefore crucial for performance. We find that all our core subroutines have a good balance between load/store, addition and multiplication instructions and can be carefully arranged to maximize instruction overlapping, achieving an IPC > 0.9. Table B.6 provides the details.

Number-theoretic transform. We implement incomplete NTTs of degrees $96 = 3 \cdot 32$ and $192 = 3 \cdot 64$ via Good's trick, using CT butterflies and Barrett multiplication throughout. Algorithm B.6 is a translation of Barrett multiplication into MVE. No explicit modular reductions are necessary during NTT and NTT^{-1}, as we confirm using a script tracking the bounds of modular representative throughout the NTT, applying Lemma 6 repeatedly.

Base multiplication. The incomplete NTTs leave us with base multiplications in rings of the form $\mathbb{Z}_q[X]/(X^4 - \zeta)$ with a $<$ 32-bit prime q, which we essentially follow from [BMK⁺21]: A polynomial $\boldsymbol{a} = a_0 + a_1X + a_2X^2 + a_3X^3$ is first expanded into a sequence $\tilde{\boldsymbol{a}} = (a_0, \ldots, a_3, \zeta a_0, \ldots, \zeta a_3)$, and 64-bit representatives of the coefficients of $\boldsymbol{a} \cdot \boldsymbol{b}$ are computed as dot products of (b_3, b_2, b_1, b_0) with length-4 subsequences of $\tilde{\boldsymbol{a}}$ with `vmalaldav`. Here, we instead compute $\tilde{\boldsymbol{a}} = \frac{1}{n}(a_0, \ldots, a_3, \zeta a_0, \ldots, \zeta a_3)$, where n is the incomplete NTT degree, taking care of the scaling by $\frac{1}{n}$ as part of the base multiplication.

CRT. We vectorize the divided-difference interpolation (cf. Equation B.1), producing chunked outputs. We allow non-reduced inputs and compute canonical reductions u_0' of u_0 and of $(u_1 - u_0')m_1$ as part of the CRT rather than at the end of NTT^{-1}. For the computation of $(u_1 - u_0')m_1$, we use refined Barrett multiplication, leveraging our choices of primes. The long multiplication $\left((u_1 - u_0')m_1 \bmod {}^{\pm}q_1\right) q_0$ is computed via `vmul` and `vmulh`, aligned to the 2^ℓ-boundary via $(a, b) \mapsto (a \bmod 2^\ell, b \cdot 2^{32-\ell} + \lfloor a/2^\ell \rfloor)$ (note $|b_i| < 2^{\lceil 52.7 \rceil - 32} = 2^{21}$, so $|2^{10}b_i| < 2^{31}$), and the low part added to u_0'. This results in a non-canonical chunked presentation of the CRT interpolation with 32-bit values, which are finally reduced to $< 2^\ell + 2^{32-\ell} = 2^{22} + 2^{10}$ via $a_i \mapsto (a_i \mod 2^\ell) + \lfloor a_{i-1}/2^\ell \rfloor$. We found that the slight relaxation of the coefficients does not impact functional correctness, while enabling vectorization of the above routine — a perfect reduction, in turn, is inherently sequential.

Constant-time lookup. In contrast to Cortex-M3, we do not use predicated move operations: A block of loads followed by a block of predicated moves allows for only very little instruction overlapping. Instead, we use load-multiply-accumulate sequences with secret constant 0/1 for the conditional moves, achieving very good instruction overlapping. Overall, we obtain a constant-time lookup of 5184 cycles for a table of 8192-bytes — 26% over the theoretical minimum of 8192/2 cycles necessary to load each table entry once with a 64-bit data path. See Appendix B.1.6.4.

As our data resides in uncached Data-TCM, it is tempting to consider a plain load for a constant time lookup. We strongly advise against this: While access to D-TCM is *typically* single-cycle, it's not in general: On Cortex-M55 a D-TCM load with secret address could happen concurrently with a DMA transfer and trigger a memory bank conflict depending on the addresses being loaded. While this particularly issue could be circumvented in our present context, it might be problematic on future micro-architectures, and it appears prudent to simply stick to the principle that memory access patterns should not rely on secret data.

B.1.5 Results

B.1.5.1 Benchmark Environment

Cortex-M3. We use the STM32 Nucleo-F207ZG with the STM32F207ZG Cortex-M3 core with 128 kB RAM and 1 MB flash. We clock the Cortex-M3 at 30 MHz (rather than the maximum frequency of 120 MHz) to void having any flash wait states when fetching code or constants from flash. We place the stack in SRAM1 (112 kB) only since it results in slightly better performance. We use libopencm3[3] and some hardware abstraction code is taken from pqm3[4]. We use the SysTick counter for benchmarking. We use `arm-none-eabi-gcc` version 11.2.0 with `-O3`.

Cortex-M55. We make use of the Arm MPS3 FPGA prototyping board with an FPGA model of the Cortex-M55r1 (AN552). Both the prototyping board and the FPGA model are publicly available[5]. Qemu supports a previous revision of the image (AN547) and can be used for running our code as well. However, for meaningful benchmarks, the FPGA board is required. We make use of the tightly coupled memory for code (ITCM) and data (DTCM). The core is clocked at the default frequency of 32 MHz. We use the PMU cycle counter for benchmarking. We use `arm-none-eabi-gcc` version 11.2.0 with `-O3`.

B.1.5.2 NTT and FNT Performance

Tables B.2 and B.3 contain the cycle counts for our core transformations. For the Cortex-M3, we implement four different transforms using specialized code for each combination of size and modulus. This allows us to minimize the number of explicit modular reductions taking into account the size of the modulus and its twiddles, and also to have a much faster FNT (modulo 65537) than the NTTs modulo 12289 and 25601. For the Cortex-M55 and a given size, the same code is used for both moduli with different precomputed constants; since no explicit modular reductions are required, we do not see prime-specific optimization potential. Base squaring and multiplication are the same, as we do not see optimization potential for squaring.

---

[3] https://github.com/libopencm3/libopencm3

[4] https://github.com/mupq/pqm3

[5] https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/download-fpga-images.

Table B.2: Performance cycles of our Cortex-M3 NTTs and FNTs in cycles.

(N, n)	$(2048, 384)$		$(4096, 768)$	
q	12289	65537	25601	65537
`NTT`	12 409	7 635	31 491	19 892
$\mathtt{NTT}_{\mathrm{heavy}}$	14 692	9 631	35 805	23 697
`basemul`	7 053	7 181	13 808	14 062
`basesqr`	6 101	6 488	11 386	12 160
$\mathtt{basemul}_{\mathrm{light}}$	5 949	5 563	11 729	10 957
$\mathtt{NTT}^{-1}$	15 130	11 090	36 227	25 015

Table B.3: Performance cycles of our Cortex-M55 NTTs and FNTs in cycles.

(N, n)	$(2048, 384)$	$(4096, 768)$
q	12289, 65537	25601, 65537
`NTT`	814	2 027
$\mathtt{NTT}_{\mathrm{heavy}}$	1 441	3 230
`basemul`	1 500	2 894
$\mathtt{basemul}_{\mathrm{light}}$	880	1 696
$\mathtt{NTT}^{-1}$	900	2 195

B.1.5.3 Modular Arithmetic: Multiplication, Squaring, Exponentiation

Table B.4 presents timings for our modular arithmetic routines, and Figure B.1 shows the distribution of cycles spent in one modular multiplication.

For Cortex-M3, we compare with BearSSL [Por] (v0.6, `i15` implementation) which to our knowledge is the only library claiming to be constant-time on the Cortex-M3. We also consider a handwritten FIOS implementation (cf. Section B.1.3.4).

On Cortex-M55, we compare to BearSSL v0.6 (`i31` implementation), to Mbed TLS [Arm] v3.1.0, and to our own handwritten FIOS implementation. The BearSSL implementation compiles down to `umlal`, while the Mbed TLS implementation uses CIOS (cf. Section B.1.3.4) with `umaal`-based inline assembly.

We find that our implementations outperform Mbed TLS and BearSSL significantly for both 2048-bit and 4096-bit parameters. Moreover, for Cortex-

M3, our NTT-based implementation is also slightly faster than the handwritten FIOS implementation for 2048-bit, and considerably faster for 4096-bit.

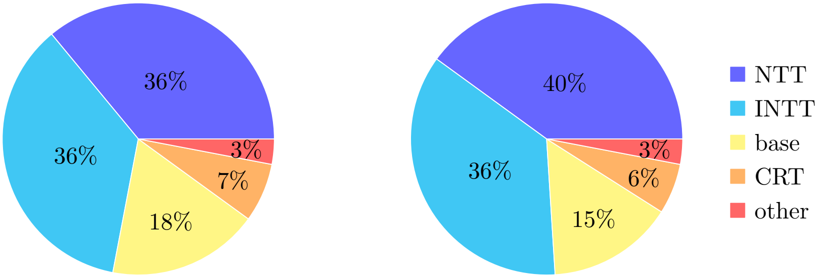
Table B.4: Performance cycles of modular multiplication, squaring, exponentiation in cycles on Cortex-M3 and Cortex-M55. $\texttt{expmod}_{\text{pub}}$ is a modular exponentiation with the exponent 65537. $\texttt{expmod}_{\text{priv}}$ is a modular exponentiation with a private n-bit exponent.

n	Work	`mulmod`	`sqrmod`	$\texttt{expmod}_{\text{pub}}$	$\texttt{expmod}_{\text{priv}}$
Cortex-M3					
2048	This work	220 047	196 830	4 227 473	494 923 435
	This work (FIOS)	234 041	–	4 912 705	543 648 872
	BearSSL [Por]	283 038	–	18 350 210	718 347 177
4096	This work	510 708	454 128	9 752 690	2 250 748 647
	This work (FIOS)	926 523	–	19 458 326	4 228 661 467
	BearSSL [Por]	1 102 151	–	70 443 207	5 505 856 187
Cortex-M55					
2048	This work	21 330	19 701	389 482	50 085 366
	This work (FIOS)	20 260	–	426 707	50 683 718
	MbedTLS [Arm]	41 443	–	884 416	108 441 240
	BearSSL [Por]	83 517	–	5 400 650	217 123 645
4096	This work	47 660	43 620	861 450	218 110 707
	This work (FIOS)	73 316	–	1 540 685	358 080 308
	MbedTLS [Arm]	152 371	–	3 223 797	755 391 521
	BearSSL [Por]	328 801	–	21 254 533	1 646 834 048

Somewhat surprisingly, the `umaal`-based handwritten FIOS is much faster than the `umaal`-based CIOS in Mbed TLS, and on par with our NTT-based implementation for 2048-bit. For 4096-bit, however, the NTT-based implementation prevails. The optimization potential between `umaal`-based FIOS and CIOS lies within memory accesses: Mbed TLS' CIOS assembly does not leverage the 64-bit data path of Cortex-M55, and merging of loops in FIOS also saves accesses. We reported this optimization potential to the Mbed TLS team.[6]

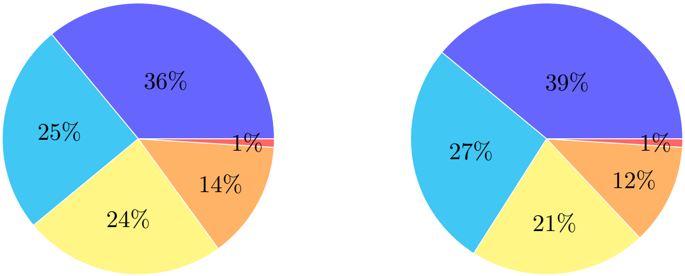
---

[6] See `https://github.com/ARMmbed/mbedtls/issues/5666` and `https://github.com/ARMmbed/mbedtls/issues/5360`.



Cortex-M3, 2048 bits

Cortex-M3, 4096 bits



Cortex-M55, 2048 bits

Cortex-M55, 4096 bits

Figure B.1: Clock cycles spent on the subroutines of a single modular multiplication.

B.1.6 Appendices

B.1.6.1 Reduction Algorithms for Cortex-M3 and Cortex-M55

Algorithm B.3 ($\log_2 \mathtt{R}$)-bit Barrett reduction on Cortex-M3.

Input: $\mathtt{a} = a$.

Output: $\mathtt{a} = a \bmod^{\pm} q$.

```
mul t, a, $\lceil \mathtt{R}/q \rfloor$
(optional) add t, t, #$(\mathtt{R}/2)$
asr t, t, #$\log_2 \mathtt{R}$
mls a, t, $q$, a
```

Algorithm B.4 16-bit Montgomery multiplication on Cortex-M3.

Input: $(\mathtt{a},\mathtt{b}) = (a, b2^{16} \bmod^{\pm} q)$.
Output: $\mathtt{a} = ab \bmod^{\pm} q$.

```
mul a, a, b
mul t, a, $-q^{-1} \bmod^{\pm} 2^{16}$
sxth t, t
mla a, t, $q$, a
asr a, a, #16
```

Algorithm B.5 FNT reduction on Cortex-M3.

Input: $\mathtt{a} = a$.
Output: $\mathtt{a} = a \bmod^{\pm} 65537 \in [-32767, 98303]$.

```
ubfx t, a, #0, #16
sub a, t, a, asr#16
```

Algorithm B.6 Barrett multiplication on Cortex-M55.

Input: $(\mathtt{a},\mathtt{b},\mathtt{b'}) = \left(a, b, \frac{\lfloor b2^{32}/q \rceil_2}{2}\right)$.
Output: $\mathtt{a} = ab \bmod^{\pm} q$.

```
vmul.s32 l, a, b
vqrdmulh.s32 h, a, b'
vmla.s32 l, h, $q$
```

B.1.6.2 On Precomputing the Montgomery Constant

Montgomery multiplication (see Section B.1.3.4) requires the precomputation of $q^{-1} \bmod \mathtt{R}$. When implementing RSA via "large" Montgomery multiplication, rather than a FIOS approach, this means that we need to precompute $n^{-1} \bmod \mathtt{R}$ for encryption and $p^{-1} \bmod \mathtt{R}$ and $q^{-1} \bmod \mathtt{R}$ for decryption. For decryption this can be computed as a part of key generation and stored as a part of the secret key. For encryption, however, it needs to be computed online.

Modular inversion $x^{-1} \bmod 2^r$ can be performed using "Hensel lifting": If $xy - 1 = 2^k a$, so that y is an inverse to x modulo 2^k, then $y' = 2y - x^2 y$ satisfies $xy' - 1 = -(xy-1)^2 = 2^{2k}a^2$, and hence y' is an inverse of x modulo 2^{2k}. This yields $x^{-1} \bmod 2^k$ after $\mathcal{O}(\log k)$ iterations. One may observe that this is the sequence of approximate solutions to $xy = 1$ for x via the Newton–Raphson method in the 2-adic integers.

We prototyped Hensel-lifting to assess its relative cost compared to the modular exponentiation; we did not seek a fully optimized version. On the Cortex-M3 we implement both a variable-time variants using `umlal` for encryption and a constant-time variant using `mla` for key generation. For the Cortex-M55, we achieve the best performance using `umaal`. We list the performance in Table B.5. We see that already a basic implementation has only a small performance overhead compared to an exponentiation (e.g., $< 5\%$ for RSA-4096).

Table B.5: Performance cycles of Hensel lifting on Cortex-M3 and Cortex-M55. Numbers for RSA-4096 in bold.

k	Cortex-M3		Cortex-M55
	`mla` (constant-time)	`umlal` (variable-time)	`umaal` (constant-time)
2112	**85 337**	45 326	**12 430**
4224	313 695	**163 107**	**38 575**

B.1.6.3 High-Level Multiplication Structure

$$\begin{tikzcd}
 & & A \in \mathbb{Z} \arrow[d, "\boxed{\texttt{chk}}"] & & & \\
 & & \mathbb{Z}[x] \arrow[d, dotted] & & & \\
 & & (\mathbb{Z}/q_1q_2)[x] \arrow[dl, dotted] \arrow[dr, dotted] & & & \\
 & (\mathbb{Z}/q_1)[x] \arrow[dl, dotted] & & (\mathbb{Z}/q_2)[x] \arrow[dr, dotted] & & \\
\mathbb{Z}[x]/(q_1, x^N - 1) \arrow[d, "\boxed{\mathsf{NTT}_1}"] & & & & \mathbb{Z}[x]/(q_2, x^N - 1) \arrow[d, "\boxed{\mathsf{NTT}_2}"] & \\
\widehat{A}_1 \in \prod_i \mathbb{Z}[x]/(q_1, x^h - \zeta_i) \arrow[d] & & & & \widehat{A}_2 \in \prod_i \mathbb{Z}[x]/(q_2, x^h - \zeta_i) \arrow[d] & \\
\odot_1 \arrow[d] & \widehat{B}_1 \arrow[l] & & & \odot_2 \arrow[d] & \widehat{B}_2 \arrow[l] \\
\widehat{C}_1 \in \prod_i \mathbb{Z}[x]/(q_1, x^h - \zeta_i) \arrow[d, "\boxed{\mathsf{NTT}_1^{-1}}"] & & & & \widehat{C}_2 \in \prod_i \mathbb{Z}[x]/(q_2, x^h - \zeta_i) \arrow[d, "\boxed{\mathsf{NTT}_2^{-1}}"] & \\
\mathbb{Z}[x]/(q_1, x^N - 1) \arrow[dr, dashed] & & & & \mathbb{Z}[x]/(q_2, x^N - 1) \arrow[dl, dashed] & \\
 & (\mathbb{Z}/q_1)[x] \arrow[dr] & & (\mathbb{Z}/q_2)[x] \arrow[dl] & & \\
 & & \boxed{\mathsf{CRT}} \arrow[d, shift left] \arrow[d, shift right] & & & \\
 & & (\mathbb{Z}/q_1q_2)[x] \arrow[d, dashed] & & & \\
 & & \mathbb{Z}[x] \arrow[d, "\boxed{\texttt{dechk}}"] & & & \\
 & & C \in \mathbb{Z} & & &
\end{tikzcd}$$

Figure B.2: High-level structure of our integer multiplication algorithm. Dotted arrows denote a conceptual reinterpretation with no change in representation, dashed arrows denote a canonical choice of lift.

B.1.6.4 Table Lookup

Algorithm B.7 Conditional move on Cortex-M3.

```
ldr.w a, [tbl, #4]
ldr.w b, [tbl, #8]
ldr.w c, [tbl, #12]
ldr.w d, [tbl], #16
cmp.n idx, #dst
itttt.n EQ
moveq.w A, a
moveq.w B, b
moveq.w C, c
moveq.w D, d
```

Algorithm B.8 Overlapping-friendly conditional accumulation on Cortex-M55.

```
cmp idx, #dst
cset mask, EQ // idx == dst
vldrw.u32 t, [tbl], #16
vmla.s32 A, t, mask
vldrw.u32 t, [tbl], #16
vmla.s32 B, t, mask
vldrw.u32 t, [tbl], #16
vmla.s32 C, t, mask
vldrw.u32 t, [tbl], #16
vmla.s32 D, t, mask
```

B.1.6.5 Pipeline Efficiency of Cortex-M55 Implementations

Table B.6 shows Performance Monitoring Unit (PMU) statistics for the subroutines of our Cortex-M55 modular exponentiation ($N = 4096$). We use `ARM_PMU_CYCCNT`, `ARM_PMU_INST_RETIRED`, `ARM_PMU_MVE_INST_RETIRED`, and `ARM_PMU_MVE_STALL` for counting cycles, retired instructions, retired MVE instructions, and MVE instructions causing a stall, respectively. We derive the rate of instructions per cycle (IPC), as well as

`ARM_PMU_MVE_INST_RETIRED/ARM_PMU_MVE_STALL` as a measure of the MVE overlapping efficiency. Despite most MVE instructions running for 2 cycles, instruction overlapping allows achieving an IPC > 0.9.

Table B.6: Performance monitoring unit statistics for Cortex-M55 implementations.

Operation	Cycle	Instruction	IPC	Inst. (MVE)	Stall	Efficiency
`NTT`	2 027	1 936	0.95	1 876	27	98.6%
$\texttt{NTT}_{\text{heavy}}$	3 231	3 017	0.93	2 742	130	96.0%
$\texttt{NTT}^{-1}$	2 195	2 128	0.96	2 072	9	99.6%
`basemul`	2 894	2 737	0.94	2 500	109	95.6%
$\texttt{basemul}_{\texttt{light}}$	1 695	1 659	0.97	1 634	6	99.6%
CRT	4 287	4 216	0.98	3 563	13	99.6%
Table lookup	5 184	4 816	0.92	4 132	12	99.7%

Appendix C

Research Data Management

This chapter gives an overview of the structure of the artifact available at the repository `https://github.com/vincentvbh/PhD_thesis_MPI-SP` and the archive `https://doi.org/10.5281/zenodo.15847923`. There are seven folders.

- `common`: This folder contains common files for each architectures/extensions. There are five subfolders: `armv7-m`, `armv7e-m`, `armv8-a`, `avx2`, and `C`. The first four subfolders, `armv7-m`, `armv7e-m`, `armv8-a`, `avx2`contain architecture-/extension-dependent files such as cryptographic hash functions, (pseudo-)random number generators, sorting functions, programs accessing cycle counters, and firmware libraries in the first two. The subfolder `C` contains C programs used in the author's research.

- `instr_bench`: This folder contains the program benchmarking some of the Armv8-A instructions.

- `macros`: This folder contains macros used in the author's research for the Armv8-A architecture.

- `matmul`: This folder contains programs demonstrating the non-one-to-one correspondence between intrinsics and assembly instructions in Armv8-A Neon.

- `mod`: This folder contains modular multiplications and compressions by the author for each architectures/extensions.

- `scheme`: This folder contains various implementations of the lattice-based cryptosystems. Aside from the implementations contributed by the author, there are also implementations by other research for comparison purpose. All of them except for the files under the subfolders `common` and files with license at the very beginning of the files, to the best of the author's knowledge, are public domain or licensed under CC0 1.0.

- `transformation`: This folder contains various implementations of the transformations. It also includes the AVX2 implementations of Kyber NTT and iNTT from the official website for benchmarking purpose.

Appendix D

Index

Summary

This thesis studies the implementation aspects of polynomial multiplications in lattice-based cryptosystems Dilithium, Kyber, Saber, NTRU, and NTRU Prime. As hardware gradually evolves and there are enormous platforms in use nowadays, systematic survey and studies with numerical justifications on the merit and the drawback of implementation techniques require a lot of effort. This thesis systematically goes through the related mathematical background, various optimization techniques and implementation aspects of polynomial multiplications, and relates them to the optimized implementations on various platforms.

Part I goes through the mathematical background. Chapter 2 reviews the basics of algebra, including rings, modules, and associative algebras, and gives some examples used throughout this thesis; Chapter 3 unifies Montgomery, Barrett, and Plantard modular arithmetic with integer approximations; Chapter 4 goes through several fast homomorphisms, including Toom–Cook, Cooley–Tukey FFT, Bruun's FFT, Good–Thomas FFT, vector-radix FFT, and Rader's FFT; Chapter 5 goes through several embedding techniques, including localization, Schönhage's FFT, Nussbaumer's FFT, and coefficient ring switching; Chapter 6 studies various optimizations related to the choices of polynomial moduli, including embedding, twisting, truncation, incomplete transformations, and Toeplitz matrix-vector products; and Chapter 7 studies various aspects of vectorization, including vectorization-friendliness, permutation-friendliness, and Toeplitz matrix-vector products.

Part II gives a general guide for optimizations. Chapter 8 reviews the platforms covered by this thesis; Chapter 9 studies the implementation aspect of modular multiplications and quotients; and Chapter 10 goes through a general guide for optimizing transformations, including layer-merging for reducing memory operations, instruction scheduling, Cooley–Tukey FFT, Good–Thomas FFT, Rader's FFT, Toom–Cook, Schönhage's FFT, and Nussbaumer's FFT.

Part III goes through several case studies and relates the mathematical techniques to platform-specific optimized implementations of each cryptosystems. Each chapter first reviews the cryptosystems, gives an overview of the polynomial arithmetic covered by this thesis and their overall optimization strategies, and goes through implementations with Armv7-M on Cortex-M3, Armv7E-M on Cortex-M4, Armv8-A on Cortex-A72 and Firestorm, and AVX2 on Haswell. Chapter 12 studies the implementations of Dilithium NTT/iNTT, matrix-vector multiplications, and challenge polynomial multiplications in Dilithium; Chapter 13 studies the implementations of Kyber NTT/iNTT, matrix-vector multiplications, inner products, and ciphertext compressions in Kyber; Chapter 14 studies the implementations of polynomial multiplications over $\mathbb{Z}_{2^k}$ and polynomial inversion in $\mathbb{Z}_2[x]/\langle \Phi_n(x) \rangle$ and $\mathbb{Z}_3[x]/\langle \Phi_n(x) \rangle$ in NTRU; Chapter 15 studies the implementations of polynomial multiplication in $\mathbb{Z}_{4591}[x]/\langle x^p - x - 1 \rangle$ and inversion in $\mathbb{Z}_3[x]/\langle x^p - x - 1 \rangle$ in the `sntrup761/ntrulpr761` NTRU Prime parameter sets; and Chapter 16 studies the implementations of matrix-vector multiplications and inner products over $\mathbb{Z}_{2^k}$ in Saber.

Samenvatting

In dit proefschrift bestuderen we de aspecten die komen kijken bij het implementeren van de cryptografische systemen Dilithium, Kyber, Saber, NTRU en NTRU Prime. Doordat processors zich geleidelijk blijven ontwikkelen, en omdat er tegenwoordig een groot scala aan processors in gebruik is, is het erg intensief om de voor-en nadelen van alle combinaties van cryptografische systemen en processors systematisch na te gaan. Dit proefschrift behandelt systematisch de gerelateerde wiskundige achtergrond, verschillende optimalisatietechnieken en implementatieaspecten van polynomiale vermenigvuldigingen, en relateert deze aan de geoptimaliseerde implementaties op verschillende platforms.

Deel !I omvat alle nodige wiskundige achtergrond. In hoofdstuk !2 bekijken we een aantal algebraïsche basisbeginselen, waaronder polynoomringen, modules en associatieve algebras; en voorbeelden van hoe deze toegepast worden in het proefschrift. In hoofdstuk !3 verenigt de Montgomery, Barrett en Plantard modulo-vermenigvuldigingsalgoritmen met integer-benaderingen. Hoofdstuk !4 omschrijft een aantal snelle homomorfismen die polynoomvermenigvuldiging implementeren, waaronder Toom–Cook, Cooley–Tukey-FFT, Bruun-FFT, Good–Thomas-FFT, vector-radix-FFT en Rader FFT. Hoofdstuk !5 beschrijft verscheidene technieken om polynoomringen te inbedden, waaronder ring-lokalisering, Schönhage's FFT, Nussbaumer's FFT en wisselen naar andere coëfficiëntringen. Hoofdstuk !6 bestudeert een aantal optimalisaties die de keuze van polynoommoduli beïnvloeden, zoals inbeddingstechnieken, "twisting", "truncation", incomplete transformaties en Toeplitz matrix-vector vermenigvuldigingen. Hoofdstuk !7 bekijkt verschillende aspecten van vectorisatie, waaronder een maat van geschiktheid (van een homomorfisme én Toeplitz matrix-vector vermenigvuldiging) voor vectoriseren en permuteren.

Deel !II biedt een algemene uiteenzetting voor het implementeren van optimalisaties. Hoofdstuk !8 geeft een overzicht van de platforms die terugkomen in de rest van het proefschrift. Hoofdstuk !9 bekijkt de implementatie-aspecten van modulovermenigvuldigingen en quotiënten. Hoofdstuk !10 geeft een uiteenzetting van geoptimaliseerde transformaties, waaronder het samenvoegen van lagen in de FFT, het schikken van instructies, Cooley–Tukey FFT, Good–Thomas FFT, Rader FFT, Toom–Cook, Schönhage FFT en Nussbaumer FFT.

Deel !III beschrijft verschillende case studies en combineert de wiskundige technieken met de platformspecifieke geöptimaliseerde implementaties van elk cryptografische systeem. Elk hoofdstuk vat eerst de cryptografische systemen samen; geeft een overzicht van de polynoomarithmetiek eerder beschreven in het proefschrift en de algemene optimalisatiestrategieën; en beschrijft implementaties voor Armv7-M op Cortex-M3, Armv7E-M op Cortex-M4, Armv8-A op Cortex-A72 en Firestorm, en AVX2 op Haswell. Hoofdstuk !11 bekijkt de implementaties van de Dilithium NTT/iNTT, matrix-vector vermenigvuldigingen, en vermenigvuldigingen van de "challenge" polynoom. Hoofdstuk !12 bekijkt implementaties van de Kyber NTT/iNTT, matrix-vector vermenigvuldigingen, inproducten en compressies van de Kyber ciphertext. Hoofdstuk !13 bekijkt implementaties van polynoomvermenigvuldigingen over $\mathbb{Z}_{2^k}$ en inversies in $\mathbb{Z}_2[x]/\langle \Phi_n(x) \rangle$ en $\mathbb{Z}_3[x]/\langle \Phi_n(x) \rangle$ in NTRU. Hoofdstuk !14 bekijkt de implementaties van polynoomvermenigvuldigingen $\mathbb{Z}_{4591}[x]/\langle x^p - x - 1 \rangle$ en inversies in $\mathbb{Z}_3[x]/\langle x^p - x - 1 \rangle$ in de `sntrup761/ntrulpr761` NTRU Prime parameter sets. Hoofdstuk !15 bekijkt the implementaties van matrix-vector vermenigvuldigingen en inproducten over $\mathbb{Z}_{2^k}$ in Saber.

總結

本論文研究晶格密碼系統 Dilithium、Kyber、Saber、NTRU、NTRU Prime 中多項式乘法之實作。因硬體的快速變革及平台的多樣性，系統性的研究及實證需要不少的心力。本論文系統性的涵蓋相關數學背景知識、各類優化技巧和多項式乘法的實作考量，並探討了各類平台上的實作。

第壹部分涵蓋相關的數學背景知識。章節貳回顧一些基礎的代數結構，包含環、模和結合代數，和一些本論文常見的例子。章節參以取整近似的觀點詮釋 Montgomery、Barrett 和 Plantard 模計算。章節肆整理了 Toom–Cook、Cooley–Tukey FFT、Bruun's FFT、Good–Thomas FFT、vector-radix FFT、Rader's FFT 各類快速同態計算。章節伍討論局部化、Schönhage's FFT、Nussbaumer's FFT 和係數環置換等各種嵌入技巧。章節陸探討基於模多項式的各種優化，包含嵌入、twisting、truncation、不完全同態變換、Toeplitz matrix-vector products。章節柒探討向量化的各個面向，包含 vectorization-friendliness、permutation-friendliness 和 Toeplitz matrix-vector products。

第貳部分整理了一些優化技巧的通則。章節捌回顧了本論文優化的目標平台。章節玖探討模乘法和商數的計算。章節拾討論了一些同態計算優化的通則，包含 layer-merging 以用於減少記憶體操作、指令排程、Cooley–Tukey FFT、Good–Thomas FFT、Rader's FFT、Toom–Cook、Schönhage's FFT 和 Nussbaumer's FFT。

第參部分探討多個案例研究及各種數學技巧在各個密碼系統中的優化。於各個章節中，我們先回顧了該章節的密碼系統和本論文涵蓋的相關多項式計算及其優化技巧。接著探討在 Cortex-M3、Cortex-M4 和 Cortex-A72 及 Firestorm 上 Armv7-M、Armv7E-M 和 Armv8-A 的實作。章節拾壹探討 Dilithium 中 Dilithium NTT/iNTT、matrix-vector multiplication 和 challenge polynomial multiplication 的實作。章節拾貳探討 Kyber 中 Kyber NTT/iNTT、matrix-vector multiplications、inner products 和 ciphertext compressions 的實作。章節拾參探討 NTRU 中係數環爲 $\mathbb{Z}_{2^k}$ 的多項式乘法和 $\mathbb{Z}_2[x]/\langle \Phi_n(x) \rangle$ 及 $\mathbb{Z}_3[x]/\langle \Phi_n(x) \rangle$ 中的反元素計算。章節拾肆探討 NTRU Prime 參數組 `sntrup761/ntrulpr761` 中 $\mathbb{Z}_{4591}[x]/\langle x^p - x - 1 \rangle$ 的多項式乘法和 $\mathbb{Z}_3[x]/\langle x^p - x - 1 \rangle$ 的反元素計算。章節拾伍探討 Saber 中係數環爲 $\mathbb{Z}_{2^k}$ 的 matrix-vector multiplications 和 inner products。

Curriculum Vitae

Vincent Hwang (黃柏文) was born in Hsinchu City (新竹市), Taiwan (臺灣) in 1997. He graduated from Kaohsiung Municipal Kaohsiung Senior High School (高雄市立高雄高級中學), Kaohsiung (高雄), Taiwan in June 2016, and was conferred a Bachelor's degree from the Department of Computer Science and Information Engineering (資訊工程學系) at National Taiwan University (國立臺灣大學), Taipei (臺北), Taiwan in June 2021 and a Master's degree from the same department in June 2022. He later joined the Max Planck Institute for Security and Privacy (Max-Planck-Institut für Sicherheit und Privatsphäre), Bochum, Germany (Deutschland) as a PhD student in January 2023.

Publications

This thesis consists of two publications during the last two years of his Bachelor's studies, three publications during his Master's studies, and six publications during his PhD studies. The other three publications during his Master's studies were already included in his Master's thesis and are excluded from this thesis, and the main part of the remaining one publication is excluded from this thesis. Below is an exhaustive list of publications in reverse chronological order by the time of the writing of this thesis.

15. Gilles Barthe, Gustavo Xavier Delerue Marinho Alves, Hugo Pacheco, José Bacelar Almeida, Luís Esquível, Manuel Barbosa, Peter Schwabe, Pierre-Yves Strub, Tiago Oliveira, and Vincent Hwang. Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3526–3544. IEEE Computer Society, 2025.

14. Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying Polynomials without Powerful Multiplication Instructions. *IACR Transactions on*

Cryptographic Hardware and Embedded Systems, 2025(1):160–202, 2024.

13. Vincent Hwang. Formal Verification of Emulated Floating-Point Arithmetic in Falcon. In *International Workshop on Security*, pages 125-141. Springer, 2024.

12. Vincent Hwang. A Survey of Polynomial Multiplications for Lattice-Based Cryptosystems. *IACR Communications in Cryptology*, 1(2), 2024.

11. Vincent Hwang. Pushing the Limit of Vectorized Polynomial Multiplication for NTRU Prime. In *Australasian Conference on Information Security and Privacy*, pages 84–102. Springer, 2024.

10. Vincent Hwang, Chi-Ting Liu, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU Prime. In *International Conference on Applied Cryptography and Network Security*, pages 24–46. Springer, 2024.

9. Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic Views of Vectorized Polynomial Multipliers – NTRU. In Anupam Chattopadhyay, Shivam Bhasin, Stjepan Picek, and Chester Rebeiro, editors, *Progress in Cryptology – INDOCRYPT 2023*, pages 177–196. Springer, 2024.

Master’s degree conferral.

---

8. Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT Multiplications for NIST-PQC KEM Lattice Finalists: Kyber, SABER, and NTRU. 2022. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022.

7. Erdem Alkim, Vincent Hwang, and Bo-Yin Yang. Multi-Parameter Support with NTTs for NTRU and NTRU Prime on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):349-371, 2022.

6. Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient Multiplication of Somewhat Small Integers using Number–Theoretic Transforms. In *International Workshop on Security*, pages 3-23. Springer, 2022.

5. Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *International Conference on Applied Cryptography and Network Security*, pages 853–871. Springer. 2022.

4. Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221-244, 2021.

3. Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):127-151, 2021.

Bachelor's degree conferral.

2. Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021.

1. Erdem Alkim, Dean Yun-Li Cheng, Chi-Ming Marvin Chung, Hülya Evkan, Leo Wei-Lun Huang, Vincent Hwang, Ching-Lin Trista Li, Ruben Niederhagen, Cheng-Jhih Shih, Julian Wälde, and Bo-Yin Yang. Polynomial Multiplication in NTRU Prime: Comparison of Optimization Strategies on Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2020.

Technical Report

In addition to the publications, there is one technical report during his PhD studies.

1. Phillip Gajland, Vincent Hwang, Jonas Janneck. Shadowfax: Combiners for Deniability. Cryptology ePrint Archive, Paper 2025/154, 2025.