Secure and Efficient Hardware Implementations of NTRU Prime

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation (Monografie)

von
Adrian Marotzke

aus
Boston, USA

2025

Gutachter:
Prof. Dr. Dieter Gollmann
Prof. Dr. Bo-Yin Yang

Tag der mündlichen Prüfung: 12. November 2024

https://orcid.org/0000-0002-5253-881X

DOI: https://doi.org/10.15480/882.15172

Abstract

Streamlined NTRU Prime is a lattice-based Key Encapsulation Mechanism (KEM) that was one of the finalists in the NIST Post-Quantum Cryptography (PQC) Standardization effort. Based on lattice assumptions, it is assumed to be secure also against attackers with access to large-scale quantum computers. Although not selected by NIST for standardization, Streamlined NTRU Prime has already seen some deployment in open-source projects. As a result, we view hardware implementations of Streamlined NTRU Prime to be of interest, especially because many of our designs are also relevant for other cryptosystems. In addition, Streamlined NTRU Prime was designed to reduce the risk of deploying a lattice-based KEM, while only incurring a low performance penalty. We investigate if this performance penalty also applies to hardware implementations and whether we can further reduce any penalty with specialized hardware designs. Our results show that the design goals of Streamlined NTRU Prime are not a barrier to highly efficient hardware implementations, and that many design choices are in fact conducive to competitive implementations.

For this, we present multiple full hardware implementations of Streamlined NTRU Prime, with two overarching flavors: High-speed, high-area implementations, and slower, low-area implementations. All of our designs are full implementations of the KEM, including all hashing and encoding, and are fully compatible with the Streamlined NTRU Prime reference implementation. We introduce several new techniques that enable high performance and efficiency, including a batch inversion for key generation, several parallel schoolbook polynomial multipliers, a high-speed radix sorting module for fixed weight sampling, a pre-hashing of shared secrets, and new en- and decoders. We implement our design on Xilinx Artix-7 and Zynq Ultrascale+ Field-Programmable Gate Arrays (FPGAs). With the high-speed design, we achieve the to-date fastest speeds for Streamlined NTRU Prime, outperforming existing full hardware implementations of other NTRU variants, as well as being competitive with state-of-the-art Kyber and Saber implementations. Our fastest high-speed design achieves a clock frequency of 400 MHz on the Zynq Ultrascale+, and has a cycle count of 52 719, 2 252 and 3 727 respectively for key generation, encapsulation and decapsulation for the sntrup653 parameter set, for a resulting latency of 131.8 µs, 5.63 µs and 9.32 µs respectively. The full, merged design consumes 7 724 slices, 46 135 LUT, 30 025 FF, 25.5 BRAM, and 26 DSP.

We also present a side-channel protected implementation of Streamlined NTRU Prime decapsulation. A significant challenge for PQC algorithms is the protection against attackers that have additional side-channel information, such as the power consumption of a device processing secret data. As a countermeasure to such attacks, masking has been shown to be a promising and effective approach. For public-key schemes, including any recent PQC schemes, usually a mixture of Boolean and arithmetic techniques is

applied on an algorithmic level. Our generic hardware implementation of Streamlined NTRU Prime decapsulation, however, follows an idea that until now was assumed to be solely applicable to symmetric cryptography: *gadget-based* masking. The logic gates of a hardware design are transformed into a secure implementation by replacing each gate with a *composable secure gadget* that operates on uniform random shares of secret values. We show the feasibility of applying this approach also to PQC schemes and present the first Public-Key Cryptography (PKC) implementation – pre- and post-quantum – masked with the gadget-based approach considering several trade-offs and design choices. We synthesize our implementation both for Artix-7 FPGAs and 45 nm Application-Specific Integrated Circuits (ASICs), yielding practically feasible results regarding area, randomness requirement, and latency. While our masked implementation does incur an overhead of a factor of approximately 13.5 in latency and 3.5 in area in comparison to our unprotected designs, this overhead is comparable to the overhead of applying gadget-based masking to symmetric ciphers. In addition, we show that the overhead could be significantly reduced with future optimizations. We formally verified the side-channel security of our implementation, as well as practically using the Test Vector Leakage Assessment (TVLA). To our knowledge, we also present the first arbitrary-order masked SHA-512 implementation in the open literature as part of our design. Finally, we analyze the applicability of our concept to the lattice-based KEM Kyber, which has been standardized by the NIST.

As an additional contribution, we describe a new blinding method to randomize and thus protect the Fujisaki-Okamoto (FO) transform, a component of many PQC KEMs, from advanced Chosen-Ciphertext Side-Channel Attacks (CC-SCAs). We also present a case study of applying the automated masking tool AGEMA to a PKC algorithm, showing that, while this approach currently leads to highly inefficient designs, future modifications of Electronic Design Automation (EDA) tools could make this approach feasible.

Keywords: NTRU Prime · Streamlined NTRU Prime · Hardware Implementation · Lattice Cryptography · Post-Quantum Cryptography · FPGA · ASIC · VHDL · Verilog · Key Encapsulation Mechanism · Masking · Higher-Order Masking · Gate-level Masking · Gadget-Based Masking · Blinding · Fujisaki-Okamoto Transform

Acknowledgements

I would first like to extend my deepest gratitude to my supervisors and reviewers, Dieter Gollmann, Bo-Yin Yang, Joppe Bos and Thomas Wille for their guidance and assistance.

Next, I am extremely grateful to the entire NTRU Prime team: Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal and Bo-Yin Yang.

I would also like to thank my colleges both at the TUHH and at NXP: Marc Gourjon, Fabian Mackenthun, Sven Hallberg, Melissa Azouaoui, Grishma Ray Pandeya, Alejandro Garza, Leonard Püttjer, Johannes Berg, Christine van Vredendaal, Björn Fey and Mustafa Ibrahim.

I am also grateful to all of my paper co-authors for the exciting times and excellent collaboration: Bo-Yuan Peng, Ming-Han Tsai, Bo-Yin Yang, Ho-Lin Chen, Grishma Ray Pandeya, Tuğrul Daim, Georg Land, Jan Richter-Brockmann and Tim Güneysu.

Special thanks also to Daniel J. Bernstein and Mohamad Kamyar Mohajerani for their help with creating some of the plots and diagrams in this thesis. I would also like to thank Jacob Applebaum for our helpful and enlightening discussions.

Finally, I would like to thank my family and friends for their unwavering support over the years.

Funding Information: This work was labelled by the EUREKA cluster PENTA and funded by German authorities under grant agreement PENTA-2018e-17004-SunRISE. This work was also supported by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany (grants 16KIS1572K, SASVI and 16KIS0658K, SysKit-HW) and by the European Commission under the grant agreement number 101070374 (CONVOLVE).

Table of Contents

List of Acronyms

AES Advanced Encryption Standard

AGEMA Automated Generation of Masked Hardware

AIG And-Inverter Graph

ANF Algebraic Normal Form

ASIC Application-Specific Integrated Circuit

BDD Binary Decision Diagram

BKZ Blockwise Korkine-Zolotarev

BRAM Block-RAM

CC-SCA Chosen-Ciphertext Side-Channel Attack

CRQC Cryptographically Relevant Quantum Computer

CRT Chinese Remainder Theorem

CVP Closest Vector Problem

DAG Direct Acyclic Graph

DFT Discrete Fourier Transformation

DLP Discrete Logarithm Problem

DRP Dual-Rail with Precharge

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processor

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie–Hellman

ECDLP Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

EDA Electronic Design Automation

EM Electromagnetic

FF Flip-Flop

FIFO First-In-First-Out

FO Fujisaki-Okamoto

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GCD Greatest Common Divisor

GE Gate Equivalent

GHPC Generic Hardware Private Circuit

HDL Hardware Description Language

HPC Hardware Private Circuit

IETF Internet Engineering Task Force

IND-CCA2 Indistinguishability under Adaptive Chosen Ciphertext Attack

KEM Key Encapsulation Mechanism

LFSR Linear Feedback Shift Register

LLL Lenstra, Lenstra, and Lovász

LNA Low Noise Amplifier

LUT Look-Up Table

LWE Learning with Error

LWR Learning with Rounding

MAC Multiply-Accumulate

MCU Microcontroller

MLWE Module Learning with Error (LWE)

MLWR Module Learning with Rounding (LWR)

NI Non-Interference

NIST National Institute of Standards and Technology

NTT Number-Theoretic Transform

OW-CPA One-way under Chosen-Plaintext Attack

PINI Probe-Isolating Non-Interference

PKC Public-Key Cryptography

PKE Public-Key Encryption

PLL Phase-Locked Loop

PQC Post-Quantum Cryptography

PRNG Pseudorandom Number Generator

RAM Random-Access Memory

RFC Requests for Comments

RLWE Ring LWE

RLWR Ring LWR

RNG Random Number Generator

ROM Read-Only Memory

RNG Random Number Generator

SCA Side-Channel Attack

SIVP Shortest Independent Vector Problem

SNI Strong Non-Interference

SoC System on Chip

SRAM Static Random Access Memory

SVP Shortest Vector Problem

TVLA Test Vector Leakage Assessment

TRNG True Random Number Generator

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WDDL Wave Dynamic Differential Logic

XOF Extendable Output Function

List of Figures

List of Algorithms

List of Tables

List of Listings

1. Introduction

The current cryptographic landscape relies heavily on algorithms that are threatened by large-scale Cryptographically Relevant Quantum Computers (CRQCs). These machines, running algorithms such as Shor's [1] and Grover's [2] would be able to break many of the ciphers used today, including RSA and those based on Elliptic Curve Cryptography (ECC). As a result, replacement algorithms have been under research for many years, under the umbrella of Post-Quantum Cryptography (PQC) [3]. These algorithms must be ready and deployed long before quantum computers become relevant because many devices are deployed in the field for decades, and process sensitive information that must remain secret for many additional years [4]. This sensitive data is under threat by an adversary that stores encrypted traffic, and then decrypts the traffic years later when a quantum computer becomes available. This is described as Mosca's Theorem and is illustrated in Figure 1.1.

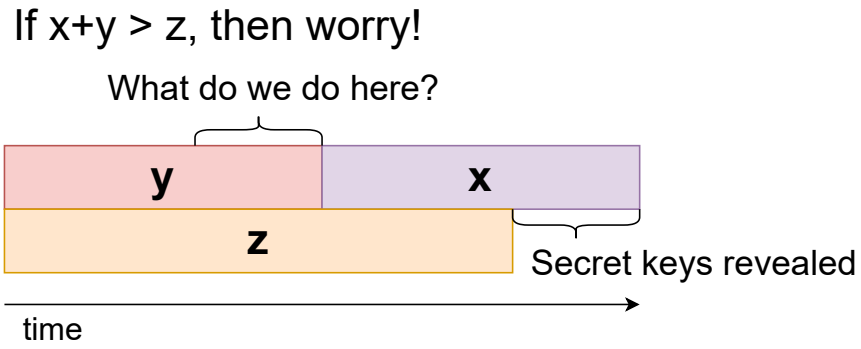


Figure 1.1.: An illustration of Mosca's Theorem [4]. z is the time until a CRQC becomes reality. y is the time to migrate to PQC. x is the time that sensitive information must remain secret.

In particular, the PQCRYPTO project [5] and the National Institute of Standards and Technology (NIST) PQC standardization project [6] have resulted in numerous new schemes designed to resist cryptanalysis by quantum computers. One common feature of all schemes is their complete dissimilarity to traditional Public-Key Cryptography (PKC) algorithms such as RSA and ECC. Instead, the schemes are based on hard problems in lattices, error correcting codes or the security of hash functions [3]. While some of these schemes are quite old and have thus been subject to a considerable amount of research concerning cryptanalysis and efficient implementations, there are also many new and cutting-edge schemes that still require additional analysis before they can be confidently deployed. Apart from security analysis, this also includes research on secure and efficient soft- and hardware implementations.

Hardware implementations of these new schemes are of significant interest because they will be needed as accelerators and co-processors in a wide range of devices, including high-end servers and small embedded devices. Such accelerators can significantly increase performance, lower power consumption and increase implementation security in comparison to pure software implementations. For example, servers and network gateways would benefit from high-speed, high-throughput hardware implementations to increase performance, while embedded devices would prefer a low-area accelerator to decrease power consumption and increase battery life. Meanwhile smartcards and other security devices require hardware implementations that are resistant to physical attackers that can exploit side-channels. All in all, this calls for fast, efficient and secure hardware implementations for a wide range of optimization targets, which allows the benchmarking and comparison of the different algorithms.

One of the comparatively "new" algorithms is NTRU Prime. NTRU Prime is a family of structured lattice Key Encapsulation Mechanism (KEM) [7] which have the goal of being secure against an attacker with a large scale CRQC. NTRU Prime is based loosely on the NTRU scheme from 1998 [8] but has several modifications to reduce the potential attack surface without sacrificing too much performance: The original paper had the subtitle "Reducing Attack Surface at Low Cost". This raises the question as to whether the design choices also only incur a low-cost performance penalty in hardware implementations, as well as whether we can reduce the performance impact with the help of specialized and carefully optimized implementations. In addition, there is also the question on how the NTRU Prime design choices impact side-channel resistant implementations.

We aim to answer these questions and close this research gap. To this end, we focus on hardware implementations of the scheme Streamlined NTRU Prime [7]. This algorithm is one of the schemes of the NTRU Prime family and is also part of the NTRU Prime submission to the NIST standardization process. From the third round onward, I was part of the submission team of NTRU Prime [9]. We will primarily use Field-Programmable Gate Arrays (FPGAs) as the implementation target throughout this thesis, but we also target Application-Specific Integrated Circuits (ASICs) for specific designs and sub-modules.

1.1. Papers & Talks

The content of this dissertation is based on contributions from the following papers, talks and code repositories:

- "NTRU Prime: Round 3 Specification", submitted to NIST in 2020 [9]. This is the submission and specification document of NTRU Prime. It is a joint work with the University of Illinois at Chicago (USA), Ruhr University Bochum (Germany), Academia Sinica (Taiwan), Tampere University (Finland), The University of Adelaide (Australia), National Taiwan University and the Technische Universiteit Eindhoven (Netherlands). I contributed to the sections on hardware implementations, as well as general proofreading.

- "A Constant-Time Full Hardware Implementation of Streamlined NTRU Prime", published and presented at the 19th International Conference of Smart Card Research and Advanced Applications (CARDIS) in 2020 [10]. The paper presents the first full hardware implementation of Streamlined NTRU Prime. A recording of the talk is available at [11], and the code of the implementation is available at [12]. The implementation in the repository is a slightly improved version, with lower cycle counts and a lower resource utilization. This is a single-author paper.
- "NTRU Prime: round-3 updates", presented by Daniel J. Bernstein at the third NIST PQC Standardization Conference in 2021 [13,14]. It presents the advantages and disadvantages of NTRU Prime, new papers and research related to NTRU Prime that occurred during the second round, as well as the updates to the submission of NTRU Prime for the third round. It is a joint work with the University of Illinois at Chicago (USA), Ruhr University Bochum (Germany), Academia Sinica (Taiwan), Tampere University (Finland), The University of Adelaide (Australia), National Taiwan University and the Technische Universiteit Eindhoven (Netherlands). I contributed to the slides on hardware implementations, as well as general proofreading.
- "A Strategy Roadmap for Post-quantum Cryptography", published in the book "Roadmapping Future: Technologies, Products and Services" in 2021 [15]. The work outlines strategy roadmaps for semiconductor vendors in the field of PQC. This is a joint work with the Northern Institute of Technology, Germany and the Portland State University, USA. Here, I acted as an adviser and contributed to the proofreading and revision.
- "PQC in the industry: The risks and challenges", an invited talk held by me at the conference "Security in the Quantum Age" in Jena, Germany in 2022 [16]. The talk discusses the new risks and challenges that PQC presents for the industry, in particular for security and semiconductor companies that will have to implement the new ciphers.
- "Streamlined NTRU Prime on FPGA", published in the Journal of Cryptographic Engineering, Springer in 2022 [17]. The paper presents a significantly improved high-end and low-area full hardware implementation of Streamlined NTRU Prime. The code of the implementation is available at [18]. This is a joint work with the National Taiwan University and the Academia Sinica, Taiwan. I contributed both schoolbook multipliers, the batch inversion, the SHA-512 module, the sorting module, the final integration into a full design, the testing and benchmarking as well as the general writing, analysis, discussion, revision and proofreading of the paper. The implementation as a whole also uses my implementation from [10] as a starting point.
- "To NTT or not to NTT: Polynomial multiplication strategies for hardware implementations of lattice cryptography", a talk held by me at the PQC Standardization & Migration Workshop collocated at Asiacrypt 2022 [19]. The talk discusses and

compares various polynomial multiplication approaches for use in lattice-based cryptography, for both FPGAs and ASICs.

- "Gadget-Based Masking of Streamlined NTRU Prime Decapsulation in Hardware", published in the Transactions on Cryptographic Hardware and Embedded Systems, Volume 2024/1, IACR [20]. The paper presents a side-channel protected hardware implementation of Streamlined NTRU Prime, using a technique called *gadget-based masking*. The code of the implementation is available at [21]. This is a joint work with the Ruhr University Bochum. I contributed to the theoretical gadget design and also implemented all masked gadgets in mixed VHDL and Verilog and integrated them into an implementation based on [17]. I also synthesized, tested and benchmarked the implementation both on FPGA and ASIC. I contributed to the general writing, analysis, discussion, revision and proofreading of the paper.

In addition to contents from the above listed papers and talks, this thesis also presents the following new contributions:

- A new blinding method to randomize the Fujisaki-Okamoto (FO) Transform, thereby reducing the effectiveness of powerful side-channel attacks against the otherwise fully deterministic FO Transform.
- A study of the use of the automatic hardware masking tool AGEMA [22] on PQC.
- A further improved high-speed hardware implementation of Streamlined NTRU Prime, reaching encapsulation and decapsulation speeds that are competitive with the fastest lattice-based schemes.

1.2. Thesis Structure

This thesis is structured as followed: First, preliminaries, technical background and prior work will be described in Part I. This includes an introduction to PQC and a detailed description of the Streamlined NTRU Prime algorithm in Chapter 2. We also describe unique design consideration for FPGAs and ASICs in Chapter 3 that influenced our hardware implementations. The background of some of the mathematical operations and algorithms we use in our implementations is described in Chapter 4. Finally, we give an overview of side-channel security in Chapter 5.

In Part II, we cover our new high-speed and low-area hardware implementations, including our works from [10] and [17]. In Chapter 6, we describe the underlying hardware modules that compute the various mathematical functions needed for Streamlined NTRU Prime, for both implementation flavors. Chapter 7 then evaluates the sub-modules and implementations for two FPGA platforms, as well as for 45 nm ASIC for specific modules. The chapter also includes a discussion on the results, a comparison with other designs from the literature and potential improvements for future work.

We present our work on side-channel resistant implementations in Part III, which includes our work in [20]. Chapter 8 describes the concept and implementation of Streamlined NTRU Prime using gadget-based masking, as well as a case study on applying the automated masking tool Automated Generation of Masked Hardware (AGEMA) to a PQC algorithm. The implementation is then evaluated for both FPGAs and 45 nm ASIC and discussed in Chapter 9. This includes improvements for future work, as well as a discussion on applying the same technique to the key generation and encapsulation of Streamlined NTRU Prime, and to the to-be-standardized PQC algorithm Kyber. Our new blinding scheme for the FO transform is described in Chapter 10.

Part I.

Preliminaries & Prior Work

2. PQC, Lattice Cryptography & NTRU Prime

In this chapter, we explain PQC and lattice cryptography in more detail, as well as introduce the NTRU Prime cryptosystem.

2.1. Post-Quantum Cryptography

The RSA cryptosystem is based on the conjectured hard problem of factoring large semiprime numbers [23]. In a similar fashion, ECC schemes such as Elliptic Curve Diffie–Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) are based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) [24, 25]. For classical computers, the best known algorithms solving these problems are either sub-exponential (RSA) [26] or exponential (ECDLP) [27].

However, in 1994 Peter Shor published a polynomial time quantum algorithm for both factoring and the Discrete Logarithm Problem (DLP), that runs on a large-scale error-correcting quantum computer [1]. Current estimates state that 10 241 logical qubits, together with $2.22 * 10^{12}$ quantum gates and a circuit depth of $1.79 * 10^{12}$ is needed for Shor's algorithm to break RSA-2048 [28]. When noisy qubits with error-correction are used, the estimates rise to 20 million qubits [29]. These estimates are still far beyond currently available quantum computers, which have about 400 qubits [30]. However due to the previously mentioned Mosca's theorem [4], it is prudent to migrate to cryptosystems that cannot be broken by quantum computers long before they become a reality.

A second quantum algorithm, discovered by Lov Grover in 1996, also impacts symmetric algorithms and hash functions [2]. Grover's algorithm offers a quadratic speedup for an unstructured search. Because a brute-force search for a symmetric key can be modeled as such an unstructured search, Grover's algorithm effectively halves the key size of symmetric algorithms: For example, an AES-128 key can be found in 2^{64} iterations. This speed-up can be efficiently countered by doubling the length of the symmetric primitives: By migrating from AES-128 to AES-256, from SHA-256 to SHA-512 and so on, the threat from Grover's algorithm can be removed.

As a response to the threat of quantum computers, the field of Post-Quantum Cryptography (PQC) began to emerge [3]. Also called quantum-resistant cryptography, the term covers cryptographic algorithms which are designed to resist cryptanalysis by quantum computers. These algorithms rely on mathematical properties other than factoring or ECDLP to achieve security. Popular families are algorithms based on coding theory, multivariate quadratic equations, isogenies of elliptic curves, hash functions and lattice

theory. In particular lattice-based algorithms have proven to be very successful, offering competitive key and ciphertext sizes as well as good performance. Lattice-based cryptography is described in more detail in Section 2.3.

2.2. NIST PQC Standardization

In 2016, NIST published a Call for Proposals of PQC algorithms for standardization [6]. A total of 69 algorithms were accepted for the first round in 2017, covering KEM, Public-Key Encryption (PKE) and signature algorithms. The algorithms and their corresponding parameter sets were placed in five different categories, called security levels, with level 1 intending to be equivalent to brute forcing AES-128, and level 5 equivalent to brute forcing AES-256. In addition to benchmarking software implementations, NIST also requested performance benchmarks of the schemes on FPGA platforms. Over the course of two further rounds, the candidates were reduced to 15 algorithms in round three. In the summer of 2022, the third round concluded [31], and four algorithms were selected as winners: the lattice-based KEM Kyber [32], the lattice-based signature schemes Falcon [33] and Dilithium [34], and the hash-based signature scheme Sphinc+ [35]. Standards for these algorithms have been published in 2024 [31, 36–38].

In addition to the selected winning algorithms, four algorithms proceeded to a fourth round. These algorithms are to be subject to additional analysis and may be standardized in the future [31]. In 2022, NIST also published a new Call for Proposals for new signature schemes [6].

2.3. Lattice-Based Cryptography

A lattice is defined as the set of points in an n-dimensional space, constructed using the integer multiples of n-linearly independent basis vectors (see Equation 2.1) [3].

$$\mathcal{L}(\mathbf{b}_1, \dots \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \right\} \tag{2.1}$$

An example for a two-dimensional lattice can be seen in Figure 2.1. The same lattice can be created by multiple different bases. A basis where the vectors are short and as orthogonal as possible is described as *good*, whereas a basis where the vectors are long and far from orthogonal is described as *bad*.

Cryptographic algorithms based on lattices rely on the presumed hardness of certain problems that exist with a lattice. The most basic problem is called the Shortest Vector Problem (SVP): Given a random basis of a lattice and some norm N, find the shortest non-zero vector in the lattice. A closely related problem to SVP is the Shortest Independent Vector Problem (SIVP): Given a random basis of a lattice with dimension n, find n linearly independent vectors $[s_1, \dots s_n]$ while minimizing $\mathsf{max}_i(||s_i||)$ according to the

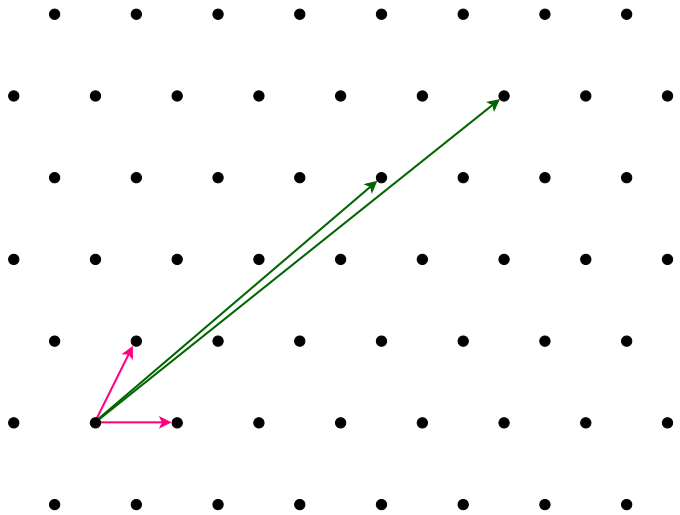


Figure 2.1.: A two-dimensional lattice, with two possible bases.

norm N. Informally SVP and SIVP can also be seen as, given a *bad* basis of a lattice, find a *good* basis [3].

Another lattice problem is the Closest Vector Problem (CVP): Given a lattice basis, a metric M to measure distance (often the Euclidean distance), and a target vector $\mathbf{t}$ that is *not* on the lattice, find the lattice point that is closest to $\mathbf{t}$. CVP has the interesting detail that while it is hard to solve given a random (i.e., a *bad*) lattice basis, it is easy if given a short enough (i.e., a *good*) basis [3].

In addition, there also exists the approximate variants of SVP, SIVP and CVP. There, the goal is to find a lattice vector whose length is at most some approximation factor $\gamma(n)$ times longer than the actually target vector in a lattice of dimension n. In practice, the approximate variants are more relevant for cryptographic algorithms based on lattice-theory because it is often sufficient to find a solution that is good enough to break the cryptoscheme [3].

A well-known and widely studied algorithm for solving the above lattice problems is the LLL algorithm, published in 1982 by Lenstra, Lenstra, and Lovász [39]. The LLL has a polynomial run-time, and an exponential approximation factor $2^{\mathcal{O}(n)}$, for lattice dimension n. When solving for polynomial approximation factors, such as with the Blockwise Korkine-Zolotarev (BKZ) algorithm, the running time is exponential [40–42]. As a result, it is conjectured that there is no polynomial time algorithm that approximates lattice problems to within polynomial factors. In addition, there is currently no known quantum algorithm that offers an exponential speedup [3]. Both of these factors have made cryptographic algorithms based on lattice problems increasingly popular,

especially for PQC.

The first cryptographic construction based on lattice problems was published in 1996 by Ajtai [43]. In 1998, the NTRU scheme was published [8], offering an efficient PKE constructions based on lattices, though without a proof reducing the NTRU problem to a known lattice problem. NTRU was subjected to a number of cryptanalysis attacks over the years, which were subsequently patched [44–46]. The Learning with Error (LWE) problem and a corresponding cryptosystem was introduced by Regev in 2005 [47], together with a proof showing that the LWE problem is asymptotically as difficult as hard problems in lattices. In 2009, Gentry presented a fully homomorphic encryption scheme based on lattices, which allows an arbitrary number of computations to be performed on encrypted data [48]. As an improvement to LWE, the Ring LWE (RLWE) cryptosystem was introduced by [49] and [50] in 2009 and 2010. Like NTRU, RLWE uses ideal lattices, also called structured lattices, to reduce the key and ciphertext sizes and increase performance. These lattices use a cyclic basis, where for lattice dimension n each of the n basis vectors $\mathbf{b}_i$ are a cyclic shift from the $i-1$ th vector, according to some polynomial ring $\mathbb{Z}[x]/f(x)$, with $f(x)$ being some degree n polynomial [49, 50].

2.4. The NTRU Prime Cryptosystem

As explained in the introduction, NTRU Prime is a family of post-quantum KEM based on structured lattices [7], loosely based on the NTRU scheme from 1998 [8]. The primary design goal of NTRU Prime was to reduce the potential attack surface, simplify the security analysis and minimize the risk of using a comparatively new lattice-based scheme while at the same time only incurring a low performance penalty. As a KEM, NTRU Prime consists of three operations:

1. *Key Generation*, where Bob generates a key pair consisting of a public key and a private key.
2. *Encapsulation*, where Alice encapsulates a random shared secret in a ciphertext using Bob's public key.
3. *Decapsulation*, where Bob decapsulates the received ciphertext from Alice using his private key to retrieve the shared secret.

This shared secret can then be used as a secret key by both parties to encrypt data with a symmetric encryption algorithm. NTRU Prime has two concrete KEMs: Streamlined NTRU Prime and NTRU LPRime. Between the two, Streamlined NTRU Prime is the recommended variant, and has smaller ciphertexts as well as faster encapsulation and decapsulation [9, 51]. As a result, this thesis will focus on Streamlined NTRU Prime.

NTRU Prime was submitted in 2017 to the NIST PQC Standardization project, as a first-round candidate [52]. In 2019 NTRU Prime was selected to continue to the second round, and several tweaks were incorporated for the start of the second round [53]. NTRU

Prime continued to the third round in 2020 as an alternate finalist, with no changes to the specification except the addition of more parameter sets [9]. Throughout this thesis, I will be using the third-round specification unless specified otherwise. In 2022, the third round concluded, and NTRU Prime was eliminated in favor of the structured lattice KEM Kyber, which was also standardization in 2024 [31, 32, 36].

Although not selected for standardization by NIST, Streamlined NTRU Prime has already received some deployment. The OpenSSH project, starting with version 9.0, uses the hybrid Streamlined NTRU Prime and the X25519 ECDH key exchange method [54] by default, for both server and client [55]. The OpenBSD project has also added Streamlined NTRU Prime, again coupled together with the X25519 ECDH key exchange [54], as an option for use in IPsec [56]. Finally, there are draft RFCs submitted to the Internet Engineering Task Force (IETF) on the standardization of Streamlined NTRU Prime [57, 58].

2.4.1. Definitions and Notations

Streamlined NTRU Prime [9] defines the following polynomial rings:

$$\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1) \tag{2.2}$$

$$\mathcal{R}/q = \mathbb{Z}/q[x]/(x^p - x - 1) \tag{2.3}$$

$$\mathcal{R}/3 = \mathbb{Z}/3[x]/(x^p - x - 1) \tag{2.4}$$

The parameters (p, q, w) of Streamlined NTRU Prime satisfy the following equations:

$$p, q \in \mathbb{P} \text{ prime} \tag{2.5}$$

$$w > 0, w \in \mathbb{Z}, 2p \geq 3w \tag{2.6}$$

$$q \geq 16w + 1 \tag{2.7}$$

$$x^p - x - 1 \text{ is irreducible in } \mathcal{R}/q \tag{2.8}$$

The recommended parameter set for Streamlined NTRU Prime is sntrup761:

$$p = 761, w = 286, q = 4591 \tag{2.9}$$

Additional parameter sets are listed in Table 2.1, as well as non-standardized parameter sets in [7]. NTRU Prime also uses the following notations:

- Small : A polynomial of $\mathcal{R}$ is small if all of its coefficients are in $\{-1, 0, 1\}$.
- Weight w : A polynomial of $\mathcal{R}$ has weight w if it has exactly w non-zero coefficients.
- Short : The set of small weight w polynomials of $\mathcal{R}$.
- Round : While viewing each coefficient as an integer between $-(q-1)/2$ and $(q-1)/2$, round all coefficients of a polynomial to the nearest multiple of 3.

- $\mathsf{Hash}_a(x)$: The SHA-512 hash of the byte array x, prepended by the single byte value a as a domain separator. Only the first 256 bits of the output hash are used.
- Encode & Decode: Streamlined NTRU Prime uses an en- and decoding algorithm to transform polynomials in $\mathcal{R}/3$ and $\mathcal{R}/q$ to and from byte strings. We use the notation of an underline to indicate that the respective value is encoded.

Table 2.1.: Streamlined NTRU Prime parameters for the NIST round 3 standardization process, together with the corresponding Core-SVP security in bits, security level, public key, private key and ciphertext sizes in bytes [9]. The Core-SVP metric is described on [59]. The public key is contained in the private key and does not have to be stored separately.

Parameter Set	Core-SVP	Level	p	q	w	Ciphertext	Public key	Private key
sntrup653	129	1	653	4621	250	897	994	1518
sntrup761	153	2	761	4591	286	1039	1158	1763
sntrup857	175	3	857	5167	322	1184	1322	1999
sntrup953	196	4	953	6343	396	1349	1505	2254
sntrup1013	209	4	1013	7177	488	1455	1623	2417
sntrup1277	270	5	1277	7879	492	1847	2067	3059

2.4.2. Algorithm Specification

The key generation of Streamlined NTRU Prime is described in Algorithm 1, and the encapsulation and decapsulation in Algorithm 2 and 3 respectively.

Algorithm 1 Streamlined NTRU Prime Key Generation [9]

repeat
  $g \xleftarrow{\$} \mathsf{Small}$ ▷ Sample a random small polynomial
until $g^{-1} \in \mathcal{R}/3$
$f \xleftarrow{\$} \mathsf{Short}$ ▷ Sample a random short polynomial
Generate a uniform random byte array ρ of length $(p+3)/4$
$h \coloneqq g/(3f) \in \mathcal{R}/q$ ▷ Compute the public key polynomial h
$\underline{K} \coloneqq \mathsf{Encode}(h)$
$\underline{k} \coloneqq \mathsf{Encode}(f, g^{-1})$
$S \coloneqq (\underline{k}, \underline{K}, \rho, \mathsf{hash}_4(\underline{K}))$
return $(S, \underline{K})$ as (private key, public key)

2.4.3. The Security of NTRU Prime

The security of Streamlined NTRU Prime is based on the so-called *NTRU problem*. It was first described in [8], and has been studied and analyzed over the decades [44, 46, 60–63]. Informally, the public key h is an element of the ring $\mathbb{Z}_\alpha[x]/\Phi$, with Φ being some degree n monic irreducible polynomial, and α an integer ≥ 2. The public key h is not uniform

Algorithm 2 Streamlined NTRU Prime Encapsulation [9]

Input Public key $\underline{K} \coloneqq \mathsf{Encode}(h)$

1: $h \in \mathcal{R}/q \coloneqq \mathsf{Decode}(\underline{K})$
2: $r \xleftarrow{\$} \mathsf{Short}$
3: $c \in \mathcal{R}/q \coloneqq \mathsf{Round}(hr)$ ▷ Compute the ciphertext polynomial c
4: $\underline{c} \coloneqq \mathsf{Encode}(c)$
5: $\underline{r} \coloneqq \mathsf{Encode}(r)$
6: $\gamma \coloneqq \mathsf{hash}_2(\mathsf{hash}_3(\underline{r}), \mathsf{hash}_4(\underline{K}))$ ▷ Compute the confirmation hash
7: $C \coloneqq (\underline{c}, \gamma)$
8: $ss \coloneqq \mathsf{hash}_1(\mathsf{hash}_3(\underline{r}), C)$ ▷ Compute the shared secret
9: **return** (C, ss) as (Ciphertext, shared secret)

random, but is computed as $h = g/f$. The polynomials $g, f \in \mathbb{Z}_\alpha[x]/\Phi$ are the secret key, and must have small magnitudes when compared to $\sqrt{\alpha}$. Two variants of the NTRU problem arise:

- The *decision variant*, which is to distinguish h from a uniform random polynomial in $\mathbb{Z}_\alpha[x]/\Phi$
- The *search variant*, which is finding a pair (g, f) from h where both f and g have sufficiently small magnitudes.

Both variants of the NTRU problem have been reduced to the SVP in lattice theory [61]. A ciphertext can be computed with $c = m + hr \in \mathbb{Z}_\alpha[x]/\Phi$, with $m, r \in \mathbb{Z}_\alpha[x]/\Phi$ also having small magnitudes. The original NTRU PKE scheme used m as the to-be-encrypted plaintext message together with a random r [8].

Streamlined NTRU Prime can be reduced to the SVP by embedding the problem of finding the private key polynomials f and g into a lattice problem [7,9], in a similar way as the original NTRU [8]. The equation

$$h = g/(3f) \in \mathcal{R}/q \tag{2.10}$$

is equivalent to

$$3hf + qk = g \in \mathcal{R} \tag{2.11}$$

for some polynomial $k \in \mathcal{R}$. This allows us to set up the following lattice equation:

$$\begin{pmatrix} k & f \end{pmatrix} \begin{pmatrix} qI & 0 \\ H & I \end{pmatrix} = \begin{pmatrix} k & f \end{pmatrix} B = \begin{pmatrix} g & f \end{pmatrix} \tag{2.12}$$

There, I is the $p \times p$ identity matrix, and H is a $p \times p$ matrix where the i'th vector is equal to $x^i \cdot 3h \bmod x^p - x - 1$, with $\cdot$ being a multiplication. H can thus be seen a cyclic shift of the public polynomial h, with each column shifted by x when compared to the previous column. The matrix B is the basis of an ideal lattice called the *Streamlined*

Algorithm 3 Streamlined NTRU Prime Decapsulation [9]

Input Ciphertext $C \coloneqq (\underline{c}, \gamma)$
Input Private key $S \coloneqq (\underline{k} \coloneqq \mathsf{Encode}(f, g^{-1}),\ \underline{K} \coloneqq \mathsf{Encode}(h),\ \rho,\ \mathsf{hash}_4(\underline{K}))$

$c \in \mathcal{R}/q \coloneqq \mathsf{Decode}(\underline{c})$
$(f, g^{-1}) \in \mathcal{R}/3 \times \mathcal{R}/3 \coloneqq \mathsf{Decode}(\underline{k})$
$h \in \mathcal{R}/q \coloneqq \mathsf{Decode}(\underline{K})$
$e \in \mathcal{R}/3 \coloneqq ((3fc) \in \mathcal{R}/q) \bmod 3$
$r' \in \mathcal{R}/3 \coloneqq g^{-1}e$
if r' does NOT have weight w **then**
    $r' \coloneqq (1, 1, \ldots, 1, 0, 0, \ldots, 0)$ ▷ The first w elements are 1, the rest 0
end if
$c' \in \mathcal{R}/q \coloneqq \mathsf{Round}(hr')$ ▷ Re-encrypt with h, r', compute new c'
$\underline{c}' \coloneqq \mathsf{Encode}(c')$
$\underline{r}' \coloneqq \mathsf{Encode}(r')$
$\gamma' \coloneqq \mathsf{hash}_2(\mathsf{hash}_3(\underline{r}'), \mathsf{hash}_4(\underline{K}))$ ▷ Re-compute the confirmation hash
$C' \coloneqq (\underline{c}', \gamma')$
if $C' = C$ **then** ▷ Compare the ciphertexts
    $ss \coloneqq \mathsf{hash}_1(\mathsf{hash}_3(\underline{r}'), C)$ ▷ Compute shared secret
    return ss
else
    $ss' \coloneqq \mathsf{hash}_0(\mathsf{hash}_3(\rho), C)$ ▷ Compute rejection
    return ss'
end if

NTRU Prime public lattice basis. This lattice has a determinant of q^p, and the vector (g, f) has a length of at most $\sqrt{2p}$. To estimate the length of the shortest vector in a random lattice, we can use Gaussian heuristics [64] which state that the shortest vector of the *Streamlined NTRU Prime public lattice* is approximately [7, 9]:

$$\mathsf{det}(B)^{(1/(2p))}\sqrt{\pi ep} = \sqrt{\pi epq} \tag{2.13}$$

This is significantly larger than $\sqrt{2p}$ for the chosen values of p and q in the Streamlined NTRU Prime parameter sets. As a result, it is highly likely that (g, f) will be the shortest nonzero vectors in the lattice. This in turn means that finding the private key polynomials (g, f) is equivalent to solving the SVP [7, 9]. A similar lattice basis can be constructed for the vector (m, r), in order to reduce the problem of finding the input r from a ciphertext to the SVP, resulting in a problem of similar difficulty.

Streamlined NTRU Prime has several differences when compared to the original NTRU scheme, which aim at reducing the attack surface and simplify the security analysis [7, 9]. Streamlined NTRU Prime uses the irreducible non-cyclotomic polynomial ring $\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1)$ together with a prime coefficient modulus q, so that the ring $\mathcal{R}/q$ is a field with large Galois group. Cyclotomic rings, for example of the form $\mathbb{Z}[x]/(x^{2^n} +$

1), are popular choices for cryptosystems due to certain implementation advantages (see also Section 4.4). However, they have also been a source of security concern [7, 65–67]. Using a non-cyclotomic ring allows Streamlined NTRU Prime to avoid certain ring homomorphisms and algebraic structures of cyclotomic polynomial rings, and the associated potential security risk. Streamlined NTRU Prime also computes a so-called *confirmation hash*, which is created by hashing the public key and the short r during an encapsulation. This ensures that a valid ciphertext can only be created by someone with a known r as input. In addition, Streamlined NTRU Prime fixes the weight of r, and rounds the ciphertext deterministically instead of adding the polynomial m as in the original NTRU PKE. This, together with certain conditions on the parameters (see Equation 2.7 and 2.6) ensures that no decryption failures can occur [7, 9]. Decryption failures have been an attack vector for several lattice-based cryptographic algorithms [45, 68–71], these attacks are systematically avoided in Streamlined NTRU Prime. In addition, because Streamlined NTRU Prime is a KEM and not a PKE, there is no need for a plaintext input [9]. A comprehensive survey of the risks avoided by Streamlined NTRU Prime can be found in [51].

Streamlined NTRU Prime also uses the Fujisaki-Okamoto Transform [72] to change a One-way under Chosen-Plaintext Attack (OW-CPA) PKE into a Indistinguishability under Adaptive Chosen Ciphertext Attack (IND-CCA2) secure KEM [72, 73]. A part of this transformation is the re-encryption of the received ciphertext during the decapsulation (see Line 9 in Algorithm 3). The new ciphertext is then compared with the original ciphertext in Line 14. Only when these match is the shared secret ss computed. Otherwise, implicit rejection is performed, where instead ss' is computed from the received ciphertext and ρ. The security properties of IND-CCA2 allow an unlimited reuse of keys [7, 9].

The subtitle of NTRU Prime "Reducing Attack Surface at Low Cost" is also in reference to other cryptosystems that incur a much larger performance penalty when reducing the attack surface. Examples for such cryptosystems are Classic McEliece [74, 75] and FrodoKEM [76, 77]. Both schemes claim to avoid many of the risks also avoided by NTRU Prime, either by using unstructured lattices (Frodo), or not using lattices at all (Classic McEliece). However, both have a much larger performance penalty than NTRU Prime, for example in the size of the public key: The public key for Frodo is in the order of ten kilobytes and around one megabyte for Classic McEliece.

3. Design Considerations for FPGAs & ASICs

FPGAs are popular hardware implementation platforms as one can easily construct and prototype customized digital logic circuits, without the large cost of manufacturing ASICs. FPGAs are programmed using a so-called Hardware Description Language (HDL). The HDL design is then synthesized with Electronic Design Automation (EDA) tools from the FPGA manufacturer, creating an image that can be flashed to the FPGA. As HDLs, we will be using both VHSIC Hardware Description Language (VHDL) and Verilog for our designs.

Most FPGAs provide several different general-purpose resources which are either common in general logic circuits or are able to simulate or execute Boolean functions. On hardware implementation with FPGAs, the utilization of these resources is one of the important standards of comparison among similar implementations. To "make an apples-to-apples comparison," a specified FPGA platform is often assigned in a call-for-proposal project. NIST recommends that "(PQC submission) teams generally focus their hardware implementation efforts on Artix-7" as an FPGA platform [78]. Artix-7 is an FPGA platform manufactured by Xilinx, now part of AMD. We will focus on Xilinx FPGAs in this thesis, in particular Xilinx Zynq Ultrascale+ and Artix-7 FPGAs as the primary target platform but note that the philosophy of the design consideration remains the same if the resources are of similar types and structures, even when the FPGA manufacturer differs. Next, we introduce the main resources provided in FPGAs.

3.1. Look-Up Tables

Look-Up Tables (LUTs) are the basic logic units in FPGAs [79, 80]. A LUT is a combinatorial logic unit with usually 4 to 6 input bits and 1 to 2 output bits. Here we denote a LUT with m input bits and n output bits as $\mathrm{LUT}_{m,n}$. A LUT can be considered as a block of read-only memory: For example, a $\mathrm{LUT}_{5,2}$ can be considered as a block with 32 cells, each of which contains 2 bit. Xilinx Zynq Ultrascale+ and Artix-7 FPGAs provide LUT units which support both the functions of $\mathrm{LUT}_{5,2}$ and $\mathrm{LUT}_{6,1}$. Usually LUT are used to implement combinatorial digital circuits, but they are also useful to implement Read-Only Memory (ROM) and Random-Access Memory (RAM). This type of RAM in FPGAs is also called distributed RAM. For example, to construct a 12 bit, 32-cell ROM unit, we will need 6 $\mathrm{LUT}_{5,2}$ units. A 13 bit, 64-cell RAM costs 13 $\mathrm{LUT}_{6,1}$ units. Several LUT are grouped together into a so-called *slice*, together with a certain number of Flip-Flops (FFs). In Artix-7 FPGAs, each slice contains four LUT, while the slices in the Xilinx Zynq Ultrascale+ contain eight LUT. As a result, a design would roughly consume half the number of slices on a Zynq Ultrascale+ FPGA compared to an Artix-7

(or older) FPGA. Slices also often contain additional hardware such as carry logic and multiplexers, as well as the necessary routing connections to make larger circuits [79,80].

3.2. Digital Signal Processor Slice

A Digital Signal Processor (DSP) slice is an arithmetic unit which consists of one multiplier and some accumulators. The multiplier supports signed integer multiplication up to a specified bit-width, and it would otherwise cost a significant amount of LUTs to construct the same multiplier if the DSP slice is not used. Xilinx Zynq Ultrascale+ FPGAs provides DSP slices with 27×18 bit signed integer multipliers [81], and Xilinx Artix-7 FPGAs provides DSP slices with 25×18 bit signed integer multipliers [82].

To multiply two integers whose bit lengths are more than the limit one DSP slice can offer, we can either apply a sequential approach, or connect two or more DSP slices in parallel. For example, to multiply a 23 bit signed integer with a 32 bit signed integer, we can connect two slices in parallel, or multiply the multiplicand with the least significant 16 bits of the other integer and then with the most significant bits. If we can control the bit lengths of the integers we want to multiply, however, we are able to limit the bit lengths so that one DSP slice can handle the multiplication.

3.3. Block-RAM

A Block-RAM (BRAM) unit can be used to store a large number of bits. Every BRAM provides several channels with partially customizable data widths during the hardware synthesis stage. We can read and/or write the data stored in one BRAM only via the channels. This implies that we can access as many words simultaneously as the number of channels in one BRAM, and if we want to access more words at the same clock cycle, we need either to duplicate the data from the BRAM to another in advance, or to partition the data we want to store in two or more BRAMs.

Both Xilinx Zynq Ultrascale+ and Artix-7 FPGAs provide BRAM units [83,84], each of which contains 36 kbit and two channels. Every BRAM unit can be divided into two blocks of 18 kbit, each of which in turn provides two channels. The synthesis report records 0.5 BRAMs of utilization if a 18 kbit block is utilized. In both FPGAs the data width of each 18 kbit block can be customized as 1, 2, 4, 9, or 18 bit.

3.4. Application-Specific Integrated Circuit

In contrast to FPGAs, ASICs have significant initial engineering cost. Depending on the exact manufacturing technology, the initial cost (e.g., of the lithography masks) can easily exceed millions of US dollars. However, once a production line has been established, ASICs have the benefit of lower unit cost, higher clock frequency and lower power consumption when compared with FPGAs [85].

Due to the high initial cost, actually manufacturing and testing an ASIC of any of the hardware implementations discussed in this thesis is infeasible. However, one can get insights on the estimated silicon area by synthesizing the HDL design to a gate-level netlist. This is done with industry standard EDA tools such as from Cadence or Synopsis, together with a so-called cell library. The cell library contains all the necessary information of a particular technology node, such as the size and layout of a NAND or XOR gate, the operating voltages, electrical switching characteristics etc. As a result, one can accurately measure the total number of gates a design would consume (called Gate Equivalent (GE)), together with the total silicon area and maximum clock frequency [85]. For this thesis, we will be using the 45nm Nangate open cell library [86], unless specified otherwise. This is an open-source cell library that contains all the elements typically needed for an ASIC, such as flip-flops, NAND gates, XOR gates and clock buffers.

4. Mathematical Operations & Auxiliary Functions

In this chapter, we explain the background details of the mathematical operations employed in NTRU Prime implementations, as well as auxiliary functions such as hashing and encoding.

4.1. Fast Polynomial Inversion

During key generation, two polynomials must be inverted, in the rings $\mathcal{R}/3$ and $\mathcal{R}/q$ respectively. This inversion must occur in constant-time (see also side-channel security in Chapter 5). A very successful and very fast inversion method is the extended Greatest Common Divisor (GCD) algorithm from [87]. It has been used in multiple implementations of Streamlined NTRU Prime, both hardware [10, 17] and software [9], NTRU [88], and code-based cryptography [89]. We will be using this method throughout this thesis. Other methods, such as Fermat's method or Hensel lifting, are not considered because they are either significantly slower, not constant-time or are not applicable to the rings used in Streamlined NTRU Prime [87, 90]. The extended GCD from [87] is based on a constant-time *division step*, also called divstep. When we apply a (sufficiently large) constant number of division steps, we can compute the GCD of two input polynomials, which in turn also allows us to also compute the inverse if the two input polynomials are coprime [87]. All of the following operations are assumed to be in $\mathcal{R}/q$, and also apply for $\mathcal{R}/3$. A division step is defined as:

$$\mathsf{divstep}(\delta, f, g) = \begin{cases} (1-\delta, g, (g(0)\cdot f - f(0)\cdot g)/x), & \text{if } \delta > 0 \text{ and } \mathrm{g}(0) \neq 0. \\ (1+\delta, f, (f(0)\cdot g - g(0)\cdot f)/x), & \text{otherwise.} \end{cases} \tag{4.1}$$

Here, f, g are the two polynomials of which we want to compute the GCD, $f(0), g(0)$ are the constant terms of the polynomials f and g respectively, and δ is computed as:

$$\delta = \text{degree of } f - \text{degree of } g \tag{4.2}$$

In order to illustrate how this can be used to compute the inversion, we shall write Equation 4.1 in matrix form. For this, we write $(\delta_1, f_1, g_1) = \mathsf{divstep}(\delta, f, g)$, i.e., (δ_1, f_1, g_1) is the output of the divstep. Then we can define:

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \tau(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix} \tag{4.3}$$

where

$$\tau(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ g(0)/x & -f(0)/x \end{pmatrix}, & \text{if } \delta > 0 \text{ and } \mathrm{g}(0) \neq 0. \\ \begin{pmatrix} 1 & 0 \\ -g(0)/x & f(0)/x \end{pmatrix}, & \text{otherwise.} \end{cases} \tag{4.4}$$

as well as

$$\begin{pmatrix} 1 \\ \delta_1 \end{pmatrix} = S(\delta, f, g) \begin{pmatrix} 1 \\ \delta \end{pmatrix} \tag{4.5}$$

where

$$S(\delta, f, g) = \begin{cases} \begin{pmatrix} 1 & 0 \\ 1 & -1 \end{pmatrix}, & \text{if } \delta > 0 \text{ and } \mathrm{g}(0) \neq 0. \\ \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, & \text{otherwise.} \end{cases} \tag{4.6}$$

Both $\tau(\delta, f, g)$ and $S(\delta, f, g)$ can be computed solely with $\delta, g(0)$ and $f(0)$ and are independent of any other polynomial coefficients. We then write the i^{th} matrix τ as

$$\tau_i = \tau(\delta_i, f_i, g_i) \tag{4.7}$$

and the i^{th} matrix S as

$$S_i = S(\delta_i, f_i, g_i) \tag{4.8}$$

We can then compute the n^{th} iteration of the τ matrix, using the δ computed with S_i:

$$\begin{pmatrix} u & v \\ r & t \end{pmatrix} = \tau_{n-1}\tau_{n-2}\ldots\tau_0 \tag{4.9}$$

If n has been chosen to be sufficiently large, and f and g are coprime, then v is the inverse of g, the rest of the variables can be discarded. In the case of Streamlined NTRU Prime, we have $n = 2 \cdot p$ [9, 87]. The C code to compute the inverse of a polynomial in $\mathcal{R}/q$ using the extended GCD algorithm can be found in Appendix A.2 in Listing A.1.

An additional detail of the inversion in Streamlined NTRU Prime is that, as can be seen in Algorithm 1, the inversion of the short polynomial f is always successful because f is nonzero and $\mathcal{R}/q$ is a field. In contrast, the inverse of the small polynomial g may not exist. In these cases, rejection sampling can be used: the non-invertible g is discarded, and a new g is sampled.

Algorithm 4 Description of Montgomery's trick for batch inversion [91].

Input n: the batch size
Input $(f_1, \ldots, f_n)$: an array of n polynomials to be inverted
Output $(f_1^{-1}, \ldots, f_n^{-1})$: the array of n inverted polynomials

1: $a_1 \coloneqq f_1$
2: **for** i from 2 to n **do**
3: $\quad a_i \coloneqq a_{i-1} \cdot f_i$
4: **end for**
5: Compute inverse a_n^{-1}
6: **for** i from n to 2 **do**
7: $\quad f_i^{-1} \coloneqq a_i^{-1} \cdot a_{i-1}$
8: $\quad a_{i-1}^{-1} \coloneqq a_i \cdot f_i$
9: **end for**
10: $f_1^{-1} \coloneqq a_1^{-1}$
11: **return** $(f_1^{-1}, \ldots, f_n^{-1})$

4.2. Montgomery's Trick for Batch Inversion

Montgomery's trick is a method to accelerate inversion by doing batch inversion [91]. This allows us to replace n inversions in a ring with a single inversion, together with $3(n-1)$ multiplications. Montgomery's trick is described in Algorithm 4. The trick can lead to a significant speedup as long as multiplication is at least 3 times as fast as a single inversion, and one has enough storage space to store the intermediate products. Batch inversion with Montgomery's trick for Streamlined NTRU Prime was already proposed in the original NTRU Prime paper [7]. It was implemented for fast key generation in an integration of Streamlined NTRU Prime into OpenSSL [90]. There, for the parameter set sntrup761 and a batch size of 32, it led to a key generation speed of 156,317 cycles per key on a Intel Xeon E3-1275v3, compared to the non-batch 819,332 cycles.

4.3. Schoolbook & Karatsuba Multiplication

Schoolbook multiplication is the simplest form of polynomial multiplication [92]. It has an asymptotic complexity of $\mathcal{O}(p^2)$, where p is the degree of the polynomial. However, it has the benefit of requiring little pre- or post-processing (aka, constant and linear factors in $\mathcal{O}$ notation), which means it can be faster than other, more complex methods for certain inputs. The schoolbook multiplication algorithm for polynomials in $\mathcal{R}/q$ can be found in Algorithm 5. It has been used previously in several hardware implementations of PQC algorithms, for example in [88, 93, 94].

An asymptotically faster method than schoolbook multiplication is Karatsuba multiplication [95]. This method involves splitting a multiplication of degree p into three partial polynomial multiplications of degree $\lceil p/2 \rceil$. The description can be found in Al-

Algorithm 5 Schoolbook polynomial multiplication [92, 93].

Input $f(x), g(x)$: the two polynomials in $\mathcal{R}/q$ of degree $p-1$ to be multiplied
Output The product of the two inputs: $f(x) \cdot g(x) \in \mathcal{R}/q$

1: $acc[0:p-1] \coloneqq 0$
2: **for** i from 0 to $p-1$ **do**
3: **for** j from 0 to $p-1$ **do**
4: $acc[j] \coloneqq acc[j] + g[j] \cdot f[i] \bmod \mathbb{Z}/q$
5: **end for**
6: $g \coloneqq g \cdot x \bmod \mathcal{R}/q$
7: **end for**
8: **return** acc

Algorithm 6 Description of the recursive Karatsuba's multiplication [95]. Once the recursion limit l is reached, schoolbook multiplication is performed. The $\mathsf{split}(x, y)$ operator separates a polynomial at position y into two parts.

Param l: the limit at which to stop recursion
Input $f(x), g(x)$: the two polynomials to be multiplied
Output The product of the two inputs: $f(x) \cdot g(x)$

1: $m \coloneqq \max(\mathsf{degree}(f(x)), \mathsf{degree}(g(x)))$
2: **if** $m \leq l$ **then**
3: **return** $f(x) \cdot g(x)$
4: **end if**
5: $m_2 \coloneqq \lfloor (m/2) \rfloor$
6: $high_1(x), low_1(x) \coloneqq \mathsf{split}(f(x), m_2)$
7: $high_2(x), low_2(x) \coloneqq \mathsf{split}(g(x), m_2)$
8: $z_0(x) \coloneqq \mathsf{Karatsuba}(low_1, low_2)$
9: $z_1(x) \coloneqq \mathsf{Karatsuba}(low_1 + high_1, low_2 + high_2)$
10: $z_2(x) \coloneqq \mathsf{Karatsuba}(high_1, high_2)$
11: **return** $z_2(x) \cdot x^{m_2 \cdot 2} + (z_1(x) - z_2(x) - z_0(x)) \cdot x^{m_2} + z_0(x)$

gorithm 6. When applied recursively, this leads to a complexity of $\mathcal{O}(p^{log_2(3)} \approx \mathcal{O}(p^{1.58})$. A downside of Karatsuba multiplication is that the recursive structure is difficult to implement in hardware, as is transforming it to an iterative structure. The same also applies to Toom-Cook multiplication [96], which is a generalization of Karatsuba's algorithm, splitting polynomials into k parts: Due to the recursive nature, it is challenging to implement in hardware. Nevertheless, both Karatsuba and Toom-Cook have been used in previous hardware implementations [88, 97, 98].

4.4. Number Theoretic Transform

The Number-Theoretic Transform (NTT) is a variant of the Discrete Fourier Transformation (DFT) [99–101]. Rather than operating on complex numbers, the NTT operates

on modular integers. The NTT can be used to efficiently multiply polynomials in the ring $\mathbb{Z}_\alpha[x]/(x^{2^n}-1)$, for the integers $n \geq 1$ and α [102,103]. The NTT requires α to be *NTT friendly*, that is, the ring $\mathbb{Z}_\alpha[x]/(x^{2^n}-1)$ must have a 2^n-th primitive root of unity ϕ to compute the so-called *twiddle factors*. The primitive root of unity is defined as:

$$\phi \in \mathbb{Z}_\alpha \text{ s.t. } \phi^{2^n} \equiv 1 \wedge \forall i < 2^n : \phi^i \neq 1 \tag{4.10}$$

The NTT polynomial multiplication can be viewed as a *cyclic convolution*, and can be computed as follows:

$$C = A \cdot B \in \mathbb{Z}_\alpha[x]/(x^{2^n}-1) \tag{4.11}$$
$$C = \mathsf{iNTT}(\mathsf{NTT}(A) \odot \mathsf{NTT}(B)) \tag{4.12}$$

The NTT can also be used to implement a polynomial multiplication for the ring $\mathbb{Z}_\alpha[x]/(x^{2^n}+1)$, which can be viewed as a *negacyclic convolution* [102,103]. For this, we require a 2^{n+1}-th root of unity Ψ:

$$\Psi \in \mathbb{Z}_\alpha \text{ s.t. } \Psi^{2^{n+1}} \equiv 1 \wedge \forall i < 2^{n+1} : \Psi^i \neq 1 \tag{4.13}$$

The 2^n-th root of unity ϕ can be computed via $\phi \equiv \Psi^2 \bmod \alpha$. The polynomial multiplication can then be performed:

$$C = A \cdot B \in \mathbb{Z}_\alpha[x]/(x^{2^n}+1) \tag{4.14}$$
$$C = \mathsf{iNTT}(\mathsf{NTT}(A \odot \vartheta) \odot \mathsf{NTT}(B \odot \vartheta)) \odot \vartheta^{-1} \tag{4.15}$$

where $\odot$ is a point-wise multiplication, and $\vartheta = (\Psi^0, \Psi^1, \ldots, \Psi^{n-1})$ and $\vartheta^{-1} = (\Psi^0, \Psi^{-1}, \ldots, \Psi^{-(n-1)})$.

The NTT polynomial multiplication has a complexity of $\mathcal{O}(p \log p)$ with regards to a polynomial of degree $p = 2^n$, which is significantly better than the $\mathcal{O}(p^2)$ or $\mathcal{O}(p^{1.58})$ of schoolbook multiplication and Karatsuba multiplication respectively. The NTT has been used in a number of implementations for lattice-based cryptography, both in hardware [17,88,102,104] and in software [103,105–107]. A description of an iterative NTT suitable for a hardware implementation is in Algorithm 7. Furthermore, algorithms such as Kyber explicitly use rings of the form $\mathbb{Z}_\alpha[x]/(x^{2^n}+1)$ with an NTT-friendly α to facilitate polynomial multiplications using the NTT [32]. However, the rings used in Streamlined NTRU Prime are $\mathcal{R}/q = \mathbb{Z}/q[x]/(x^p - x - 1)$ and $\mathcal{R}/3 = \mathbb{Z}/3[x]/(x^p - x - 1)$ and not $\mathbb{Z}_\alpha[x]/(x^{2^n}+1)$. As a result, we cannot apply the NTT polynomial multiplication as described in Equations 4.14 and 4.15. In order to compute polynomial multiplication in the rings used by Streamlined NTRU Prime using the NTT, additional tricks are needed. Several of such tricks are described in [106], we will describe two of them in the following sections: Good's Trick and the Chinese Remainder Theorem (CRT). We shall use the parameter set sntrup761 as an example in the following sections.

Algorithm 7 In-place iterative Number-Theoretic Transform from [103, 108], based on the Gentleman-Sande butterfly pattern. An important detail to remember is that the output $\mathsf{NTT}(A(x))$ is indexed in bit-reversed order.

Input n: Polynomial degree is $2^n - 1$
Input α: Integer modulus, such that a primitive 2^n-th root of unity exists in $\mathbb{Z}_\alpha$
Input $\phi \in \mathbb{Z}_\alpha$ s.t. $\phi^{2^n} \equiv 1 \wedge \forall i < 2^n : \phi^i \neq 1$: primitive 2^n-th root of unity
Input $\Phi[i] \coloneqq \phi^i \ \forall i < 2^n$: The powers of ϕ, called twiddle factors
Input $A(x) = \sum_{i=0}^{p-1} A_i x^i \in \mathbb{Z}_\alpha[x]/(x^{2^n} + 1)$: Coefficients of input polynomial
Output $\mathsf{NTT}(A(x))$: In-place evaluation of input $A(x)$ at the powers of the ϕ, i.e., the spectral coefficients of $A(x)$. This is also called as $A(x)$ in the NTT domain.

for *stage* from 1 to n **do**
  $\mathit{offset} \coloneqq 2^{n-\mathit{stage}}$
  for *bracket* from 0 to $2^{\mathit{stage}-1} - 1$ **do**
    for *cross* from 0 to $\mathit{offset} - 1$ **do**
      $i_E \coloneqq 2 \cdot \mathit{bracket} \cdot \mathit{offset} + \mathit{cross}$ ▷ Index calculation
      $i_O \coloneqq 2 \cdot \mathit{bracket} \cdot \mathit{offset} + \mathit{cross} + \mathit{offset}$
      $i_\phi \coloneqq 2^{\mathit{stage}-1} \cdot \mathit{cross}$
      $U \coloneqq A_{i_E}$ ▷ Copy coefficients into temporary variable
      $V \coloneqq A_{i_O}$
      $W \coloneqq \Phi[i_\phi]$
      $E \coloneqq (U + V) \bmod \alpha$ ▷ Computation
      $O \coloneqq (U - V) \cdot W \bmod \alpha$
      $A_{i_E} \coloneqq E$ ▷ Write back
      $A_{i_O} \coloneqq 0$
    end for
  end for
end for

4.4.1. Multiplication using Good's Trick and the NTT

Polynomials in $\mathcal{R}/q$ can be written as

$$f(x) = \sum_{i=0}^{p-1} f_i x^i \tag{4.16}$$

where $-\frac{q-1}{2} \leq f_i \leq \frac{q-1}{2}$ for every i satisfying $0 \leq i \leq p-1$. The polynomial multiplication of two polynomial $f(x)$ and $g(x)$ in $\mathcal{R}/q$ is

$$h(x) \triangleq f(x) \cdot g(x) \triangleq (f(x)g(x) \bmod^{\pm} q) \bmod x^p - x - 1 \tag{4.17}$$

where we denote $r = n \bmod^{\pm} q$ (signed modulo) for any integer n and r if $-\frac{q-1}{2} \leq r \leq \frac{q-1}{2}$ and an integer m exists such that $n = mq + r$. The reduction modulo $x^p - x - 1$ is easy, as we only need to substitute x^j with $x^{j-p+1} + x^{j-p}$ for every $j \geq p$ and reduce the eventual polynomial into the form $\sum_{i=0}^{p-1} h_i x^i$. As a result, the key is to evaluate $f(x)g(x) \bmod^{\pm} q$.

One method to use the NTT to compute $f(x)g(x) \bmod^{\pm} q$ is to use an NTT of sufficient size such that no polynomial reduction occurs [102,103]. For this, the input polynomials are zero-padded, and the polynomial reduction is performed manually after the NTT is complete. The NTT is a 2^n-point transformation method with a pre-determined positive integer n (written as $\mathsf{NTT}_{2^n}(\cdot)$, and the inverse operation $\mathsf{iNTT}_{2^n}(\cdot)$). For polynomials $f(x)$ and $g(x)$ of degree at most 2^n-1 that are padded with at least 2^{n-1} zero coefficients, we can implement the polynomial multiplication as

$$\begin{aligned} f(x)g(x) &= f(x)g(x) \bmod x^{2^n} - 1 \\ &= \mathsf{iNTT}_{2^n}(\mathsf{NTT}_{2^n}(f(x)) \odot \mathsf{NTT}_{2^n}(g(x))) \end{aligned} \tag{4.18}$$

For Streamlined NTRU Prime and the parameter set sntrup761 with $p = 761$, we need to pad 263 monomials with zero coefficients to the polynomials, allowing us to use $\mathsf{NTT}_{2^{11}}(\cdot)$. Because we perform the cyclic convolution, we do not need the point-wise multiplication with ϑ and ϑ^{-1} and thus only need a primitive 2^{11}-th root of unity ϕ. The downside of this approach is the relatively large amount of padding we have to perform, which consumes additional memory and forces us to use a much larger NTT [17,102,103].

Good [109] provides another approach, applying $\mathrm{NTT}_{2^9}(\cdot)$ instead and then doing three degree-512 polynomial multiplications. This idea was proposed for NTRU Prime originally in [105] and [106]. For the parameter set sntrup761 and $p = 761$, we regard the polynomial as of degree 767 instead, with the coefficients of the high-degree terms set to 0. We thus only need to pad 7 zeros to the polynomials. Now since $f(x)g(x)$ is at most of degree 1534, we have $f(x)g(x) = f(x)g(x) \bmod x^{3\cdot 512} - 1$. We set $x = yz$ and show that:

$$f(x)g(x) = f(yz)g(yz) \bmod (y^3 - 1)(z^{512} - 1) \tag{4.19}$$

We can see that for the set of integers i in $[0, 1535]$, the mapping

$$i \equiv 513\ell + 512j \pmod{1536} \tag{4.20}$$

to the set of the integer pairs (j, ℓ) where $0 \leq j \leq 2$ and $0 \leq \ell \leq 511$ is a one-to-one mapping. Then $f(x)$ (as well as $g(x)$) can be expressed as

$$\begin{aligned} f(x) &= \sum_{i=0}^{760} f_i x^i + \sum_{i=761}^{1535} 0x^i \equiv \sum_{i=0}^{1535} f_i y^{i \bmod 3} z^{i \bmod 512} \\ &= \sum_{j=0}^{2} \sum_{\ell=0}^{511} y^j z^\ell f_{(513\ell+512j) \bmod 1536} \\ &\equiv (\sum_{\ell=0}^{511} f_{513\ell \bmod 1536} \cdot z^\ell) + (\sum_{\ell=0}^{511} f_{(513\ell+1024) \bmod 1536} \cdot z^\ell)y \\ &\quad + (\sum_{\ell=0}^{511} f_{(513\ell+512) \bmod 1536} \cdot z^\ell)y^2 \end{aligned}$$

$$\equiv (\sum_{\ell=0}^{511} f_{(\ell \bmod 3)2^9+\ell} \cdot z^\ell) + (\sum_{\ell=0}^{511} f_{((\ell-1) \bmod 3)2^9+\ell} \cdot z^\ell)y + (\sum_{\ell=0}^{511} f_{((\ell-2) \bmod 3)2^9+\ell} \cdot z^\ell)y^2 \pmod{(y^3-1)(z^{512}-1)} \tag{4.21}$$

This permutation of the coefficients while splitting $f(x)$ into three polynomials is known as Good's permutation [106, 109]. For convenience we define

$$f_{y^j}(z) \triangleq \sum_{\ell=0}^{511} f_{((\ell-j) \bmod 3)2^9+\ell} \cdot z^\ell \tag{4.22}$$

with $j \in \{0, 1, 2\}$ as well as

$$f(x) \equiv f_{y^0}(z) + f_{y^1}(z)y + f_{y^2}(z)y^2 \pmod{(y^3-1)(z^{512}-1)} \tag{4.23}$$

We also define the equivalent for $g(x)$. We can assert that $f_{y^j}(z)$ and $g_{y^j}(z)$ for $j \in \{0, 1, 2\}$ all have a z-degree of 511 and also have at least half of the coefficients equal to zero. Because of this, we can evaluate $f_{y^j}(z)g_{y^j}(z)$ as

$$h_{y^j}(z) = f_{y^j}(z)g_{y^j}(z) \equiv \mathrm{iNTT}_{2^9}(\mathrm{NTT}_{2^9}(f_{y^j}(z)) \odot \mathrm{NTT}_{2^9}(g_{y^j}(z))) \tag{4.24}$$

Then $f(x)g(x)$ is given by

$$\begin{aligned} h(x) = f(x)g(x) &\equiv (f_{y^0}(z) + f_{y^1}(z)y + f_{y^2}(z)y^2)(g_{y^0}(z) + g_{y^1}(z)y + g_{y^2}(z)y^2) \\ &\equiv (f_{y^0}(z)g_{y^0}(z) + f_{y^1}(z)g_{y^2}(z) + f_{y^2}(z)g_{y^1}(z)) \\ &\quad + (f_{y^0}(z)g_{y^1}(z) + f_{y^1}(z)g_{y^0}(z) + f_{y^2}(z)g_{y^2}(z))y \\ &\quad + (f_{y^0}(z)g_{y^2}(z) + f_{y^1}(z)g_{y^1}(z) + f_{y^2}(z)g_{y^0}(z))y^2 \\ &\triangleq h_{y^0}(z) + h_{y^1}(z)y + h_{y^2}(z)y^2 \\ &\triangleq h(z, y) = \sum_{j=0}^{2} \sum_{\ell=0}^{511} h_{j\ell} z^\ell y^j \pmod{(y^3-1)(z^{512}-1)} \end{aligned} \tag{4.25}$$

We can regard the polynomial multiplication of $h(x) = f(x)g(x)$ as a schoolbook multiplication with respect to y, where the coefficients of the powers of y's are the sum of products of the polynomials in z, which can be computed using the NTT. In Equation 4.25 we can see that for every $h_{j\ell}$, the index j directs to the coefficient polynomial of y^j, and the index ℓ directs to the coefficient of z^ℓ in each polynomial. To map back the coefficients of $h(z, y)$ to those of $h(x)$, we can again apply Good's permutation:

$$h(x) = \sum_{i=0}^{1535} h_{(i \bmod 3),(i \bmod 512)} x^i \tag{4.26}$$

4.4.2. Chinese Remainder Theorem and the NTT

Although Good's Trick allows us to multiply polynomial in $\mathcal{R}/q$ using the NTT, it does not address the requirements for roots of unity: To compute the $\text{NTT}_{2^n}(\cdot)$ we need to find a 2^n-th primitive root of unity in the field $\mathbb{Z}/q$. Specifically, to apply Good's trick for $p = 761$ and $q = 4591$, we need to find a 512-th root of unity in $\mathbb{Z}/4591$. Unfortunately, such a root of unity does not exist. In fact, such roots of unity do not exist for any Streamlined NTRU Prime parameter set. One alternative is to use a sufficiently large NTT-friendly modulus (that has the necessary roots of unity) so that no overflow occurs. After the NTT is complete, we can then manually reduce back to the original modulus. However, this has the downside of requiring significantly larger multipliers, as well as needing more memory. A method to circumvent this issue is to use the Chinese Remainder Theorem (CRT), replacing the single large prime modulus with several smaller ones. Applying this to Streamlined NTRU Prime was suggested in [106]. To illustrate how the CRT can be applied, we consider the following two cases:

Case 1: The polynomial multiplications used in the standard of Streamlined NTRU Prime are multiplications with one $\mathcal{R}/3$ polynomial (coefficients are all -1, 0, or 1) and one $\mathcal{R}/q$ polynomial (coefficients are in the range $[-\frac{q-1}{2}, \frac{q-1}{2}]$. If we use the schoolbook scheme, we can see that all of the coefficients in the polynomial multiplication are in the range of $[-\frac{p(q-1)}{2}, \frac{p(q-1)}{2}]$ if we don't apply the modulo q. This is a 22 bit signed integer for the sntrup761 parameter set. If instead we want to apply Good's trick and use an NTT, we can choose two NTT-friendly primes (in whose finite fields we can find a 512-th root of unity) q_1 and q_2 such that $q_1 q_2 > p(q-1)+1$. Then we apply Good's trick from Section 4.4.1 separately twice, once with each prime. We then have two output polynomials, $h_1(x)$ and $h_2(x)$.

$$h_1(x) = \sum_{i=0}^{p-1} h_{i,1} x^i \qquad h_{i,1} \in \mathbb{Z}/q_1 \tag{4.27}$$

$$h_2(x) = \sum_{i=0}^{p-1} h_{i,2} x^i \qquad h_{i,2} \in \mathbb{Z}/q_2 \tag{4.28}$$

From the coefficients of $h_{i,1}$ and $h_{i,2}$ in $\mathbb{Z}/q_1$ and $\mathbb{Z}/q_2$ respectively, we can get the final coefficients h_i by computing

$$h_{i,1} q_2' q_2 + h_{i,2} q_1' q_1 \equiv ((h_{i,1} q_2') \bmod^{\pm} q_1) q_2 + ((h_{i,2} q_1') \bmod^{\pm} q_2) q_1 \triangleq \hat{h}_i \tag{4.29}$$

where

$$q_1' \equiv q_1^{-1} \pmod{q_2} \tag{4.30}$$

$$q_2' \equiv q_2^{-1} \pmod{q_1} \tag{4.31}$$

We can see that $\hat{h}_i$ is in the range $[-q_1q_2 + \frac{q_1+q_2}{2}, q_1q_2 - \frac{q_1+q_2}{2}]$. We only need to check if it is in the range $[-\frac{q_1q_2}{2}, \frac{q_1q_2}{2}]$ and adjust up or down by q_1q_2 if necessary i.e.,

$$\check{h}_i \triangleq \hat{h}_i + kq_1q_2, k \in \{-1, 0, 1\} \tag{4.32}$$

$$h_i \equiv \check{h}_i \pmod^{\pm} q) \tag{4.33}$$

$$h(x) = \sum_{i=0}^{p-1} h_i x^i \in \mathcal{R}/q \tag{4.34}$$

Note that we can control the logic and the primes q_1, q_2 such that we always multiply a 25 bit signed integer with a 18 bit signed integer. This is beneficial as the multipliers in the DSP slices of Xilinx FPGAs all support at least 25×18 bit signed multiplication. Controlling the size of the multiplication in this manner provides portability between high-end and low-end FPGAs and utilizes the built-in multipliers with a higher effectiveness [17]. This detail is important for the next case.

Case 2: In our implementation we apply batch inversion (see Section 4.2 and 6.6), which requires multiplication of two $\mathcal{R}/q$ polynomials. In this case, all coefficients of the polynomial multiplication result prior to applying modulo q are in the range of $[-\frac{p(q-1)^2}{4}, \frac{p(q-1)^2}{4}]$, which is $[-4008206025, 4008206025]$ for the parameter set `sntrup761`, which in turn means that the coefficients are 33 bit signed integers. As a result, we pick three NTT-friendly primes q_1, q_2, q_3. We then have three output polynomials, $h_1(x)$, $h_2(x)$, and $h_3(x)$, with the respective coefficients $h_{i,1}$, $h_{i,2}$, and $h_{i,3}$. In a similar way as in the first case, we can compute the final coefficients h_i:

$$\begin{aligned} h_{i,1}q'_{23}q_2q_3 + h_{i,2}q'_{31}q_3q_1 + h_{i,3}q'_{12}q_1q_2 &\equiv ((h_{i,1}q'_{23}) \bmod^{\pm} q_1)q_2q_3 \\ &\quad + ((h_{i,2}q'_{31}) \bmod^{\pm} q_2)q_3q_1 \\ &\quad + ((h_{i,3}q'_{12}) \bmod^{\pm} q_3)q_1q_2 \\ &\triangleq \hat{h}_i \end{aligned} \tag{4.35}$$

where

$$q'_{23} \equiv (q_2q_3)^{-1} \pmod{q_1} \tag{4.36}$$

$$q'_{31} \equiv (q_3q_1)^{-1} \pmod{q_2} \tag{4.37}$$

$$q'_{12} \equiv (q_1q_2)^{-1} \pmod{q_3} \tag{4.38}$$

We again know that $\hat{h}_i$ is in the range of in $[-\frac{3q_1q_2q_3}{2}, \frac{3q_1q_2q_3}{2}]$. We can then check if it is in the range $[-\frac{q_1q_2q_3}{2}, \frac{q_1q_2q_3}{2}]$ and adjust up or down by $q_1q_2q_3$ if necessary i.e.,

$$\check{h}_i \triangleq \hat{h}_i + kq_1q_2q_3, k \in \{-1, 0, 1\} \tag{4.39}$$

$$h_i \equiv \check{h}_i (\bmod^{\pm} q) \tag{4.40}$$

For the parameter set sntrup761, we choose $q_1 = 7681$, $q_2 = 12289$ and $q_3 = 15361$. In this case, $q'_{23} = 2562 = (\mathrm{A02})_{16}$, $q'_{31} = 8182 = 2^{13} - (\mathrm{A})_{16}$ and $q'_{12} = 10 = (\mathrm{A})_{16}$. This means that all $h'_{i,a} = (h_{i,a} q'_{bc}) \bmod^{\pm} q_a$ computations can be done with comparatively simple bit shifts, additions, and subtractions followed by a modulo operation. The choice of primes also ensures that all $h'_{i,a}$ are represented as 14 bit signed integers. Multiplying the remaining $q_b q_c$ can be also done by one 25×18 bit multiplier since in this configuration:

$$\begin{aligned}
&h'_{i,q_1} q_2 q_3 + h'_{i,q_2} q_3 q_1 + h'_{i,q_3} q_1 q_2 \\
&= h'_{i,q_1} \cdot 188771329 + h'_{i,q_2} \cdot 117987841 + h'_{i,q_3} \cdot 94391809 \\
&= h'_{i,q_1}(184347 \cdot 2^{10} + 1) + h'_{i,q_2}(230445 \cdot 2^9 + 1) + h'_{i,q_3}(184359 \cdot 2^9 + 1) \\
&= h'_{i,q_1}((\mathrm{2D01B})_{16} \cdot 2^{10} + 1) + h'_{i,q_2}((\mathrm{3842D})_{16} \cdot 2^9 + 1) \\
&+ h'_{i,q_3}((\mathrm{2D027})_{16} \cdot 2^9 + 1)
\end{aligned} \tag{4.41}$$

Ultimately, this enables all multiplication to be implemented in an individual DSP slice [17].

4.5. Encode and Decode Algorithm

Streamlined NTRU Prime uses multiple encoding and decoding algorithms for converting polynomials to and from byte strings [9]. For the en- and decoding of small polynomials in $\mathcal{R}/3$, each coefficient can be stored in 2 bits. The encoding is thus very simple: Four coefficients are stored in one byte. Each small polynomial can be stored as a byte string of length $\lceil p/4 \rceil$.

For the polynomials in $\mathcal{R}/q$, a more complex general-purpose encoder and decoder is used [9]. The codec uses the fact that coefficient values in the interval between q and 2^{13} do not occur. In addition, ciphertexts coefficients are all rounded to a multiple of 3, which again means that certain values do not occur. This can be used to save space by "overlapping" the coefficients. The python code for the encoder can be found in Listing 4.1, and the code for the decoder in Listing 4.2. For the encode algorithm, the coefficients are stored in the array R. The array M is an array of length p filled with q's when encoding a public key, and the value $(q-1)/3+1$ when encoding ciphertexts. The rounded ciphertexts allows the encoding to be slightly more efficient. For the decoder, the byte string is given as the input S, and the input M is equivalent as with the encoder.

4.6. Secure Hash Algorithm

Streamlined NTRU Prime internally uses the SHA-512 hash function. SHA-512 employs a Merkle-Damgård construction processing a 512 bit state divided into eight 64 bit words A, B, C, D, E, F, G, H [110]. In order to update the state, SHA-512 uses a round function (see Figure 4.1). The round function consists of seven adders (modulo 2^{64}), the two

```
limit = 16384
def Encode(R,M):
    if len(M) == 0: return []
    S = []
    if len(M) == 1:
        r,m = R[0],M[0]
        while m > 1:
            S += [r%256]
            r,m = r//256,(m+255)//256
        return S
    R2,M2 = [],[]
    for i in range(0,len(M)-1,2):
        m,r = M[i]*M[i+1],R[i]+M[i]*R[i+1]
        while m >= limit:
            S += [r%256]
            r,m = r//256,(m+255)//256
        R2 += [r]
        M2 += [m]
    if len(M)&1:
        R2 += [R[-1]]; M2 += [M[-1]]
    return S+Encode(R2,M2)
```

Listing 4.1: The Python code of the encoder for polynomials in $\mathcal{R}/q$ [9]. The lists R and M must have the same length, and $\forall i : 0 \leq R[i] \leq M[i] \leq 2^{14}$. When this is the case, then $\mathsf{Decode}(\mathsf{Encode}(R; M); M) = R$.

```
limit = 16384
def Decode(S,M):
    if len(M) == 0: return []
    if len(M) == 1:
        return [sum(S[i]*256**i for i in range(len(S)))%M[0]]
    k = 0; bottom,M2 = [],[]
    for i in range(0,len(M)-1,2):
        m,r,t = M[i]*M[i+1],0,1
        while m >= limit:
            r,t,k,m = r+S[k]*t,t*256,k+1,(m+255)//256
        bottom += [(r,t)]
        M2 += [m]
    if len(M)&1:
        M2 += [M[-1]]
    R2 = Decode(S[k:],M2)
    R = []
    for i in range(0,len(M)-1,2):
        r,t = bottom[i//2]; r += t*R2[i//2];
        R += [r%M[i]]; R += [(r//M[i])%M[i+1]]
    if len(M)&1:
        R += [R2[-1]]
    return R
```

Listing 4.2: The Python code of the decoder for polynomials in $\mathcal{R}/q$ [9].

functions Σ_0 and Σ_1, and the functions SHA-Ch and SHA-Ma. The former two functions Σ_0 and Σ_1 consist of simple rotation operations (denoted as $\ggg$) by three different values for each function processing A and E, respectively. The outputs of the shifts are added together by XOR operations. SHA-Ch and SHA-Ma are both non-linear function processing E, F, G and A, B, C, respectively.

$$\text{SHA-Ch}(E, F, G) = (E \wedge F) \oplus (\overline{E} \wedge G) \tag{4.42}$$
$$\text{SHA-Ma}(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \tag{4.43}$$
$$\Sigma_0(A) = (A \ggg 28) \oplus (A \ggg 34) \oplus (A \ggg 39) \tag{4.44}$$
$$\Sigma_1(E) = (E \ggg 14) \oplus (E \ggg 18) \oplus (E \ggg 41) \tag{4.45}$$

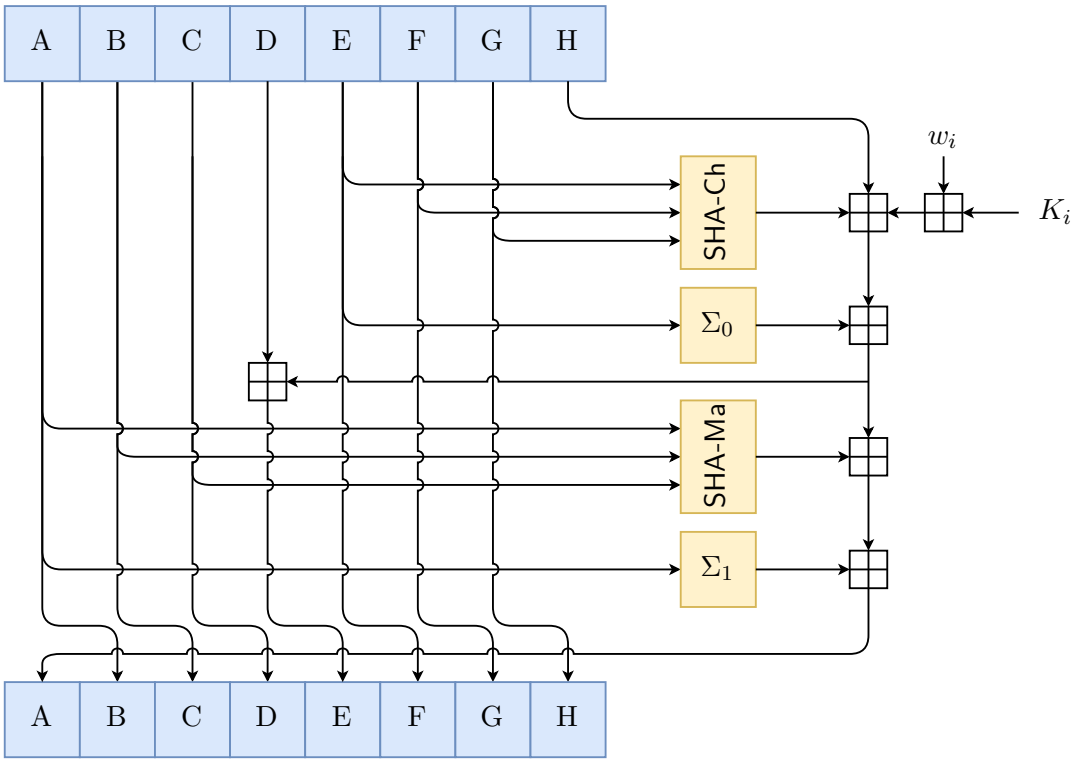


Figure 4.1.: Round function of the SHA-512 hash function. Addition modulo 2^{64} is denoted as $\boxplus$ and i is the round counter. K_i are fixed round constants, computed from the fractional parts of the cube roots of prime numbers [110].

SHA-512 operates on inputs blocks of 1 024 bit. Longer inputs are divided into 1 024 bit chunks and processed successively. The final block is padded:

1. First, a single '1' bit is appended.

2. Then, M '0' bits are appended, such that $M \geq 0$ and $L+1+M+128$ is a multiple of $1\,024$, where L is the length of the original input message in bits.
3. Finally, L is appended as a 128 bit big-endian integer.

The input block is prepossessed into a message schedule array w_i of 64 bit words, for $0 \leq i \leq 79$. The $1\,024$ bit input is split into sixteen 64 bit words and copied into w_0 to w_{15}. The rest of the array is then computed:

$$S_i = (w_{i-15} \ggg 7) \oplus (w_{i-15} \ggg 18) \oplus (w_{i-15} \gg 3) \tag{4.46}$$

$$R_i = (w_{i-2} \ggg 17) \oplus (w_{i-2} \ggg 19) \oplus (w_{i-12} \gg 10) \tag{4.47}$$

$$w_i = w_{i-16} + S_i + w_{i-7} + R_i \tag{4.48}$$

Here, $\gg$ denotes the shift operation. SHA-512 applies the round function a total of 80 times per input block [110].

5. Side-Channel Security

5.1. Side-Channel Attacks

Real-world cryptographic implementations are not the black boxes often described in cryptographic models, where only inputs and outputs such as the plaintext or ciphertext are visible. Instead, real-world implementation can leak additional data to physical attackers, such as timing or power consumption information. These type of attacks are called Side-Channel Attacks (SCAs), and it is vital for deployed implementations processing sensitive data to also provide security against such physical attacks. For implementations on server machines and personal computers, it usually suffices to use strictly constant-time implementations by means of having an execution time independent of secret values. This includes not performing time-variable branching based on secret data, as well not loading values from secret addresses in memory systems that use a cache [111–113]. An example for such a timing attack on a PQC scheme is presented in [114], where they attack the re-encapsulation of FrodoKEM because the ciphertext comparison was not constant-time.

For embedded devices, however, we additionally have to consider adversaries who can measure the power consumption or Electromagnetic (EM) emanation of a device processing secret data [115–117]. In this context, many practical attacks have been shown in the past on "classical", but also on PQC schemes. For instance, several attacks have been published attacking Kyber [118–121], Saber [122], Falcon [123], NTRU [124], or even generic lattice-based constructions [125]. Notable here are attacks targeting *side-channel protected* implementations, such as the attacks on supposedly protected Kyber [126, 127] and Saber implementations [128, 129].

Specifically for Streamlined NTRU Prime, two attacks have been proposed. First, Xu et *al.* show single-trace attacks on fixed weight sampling as used in Streamlined NTRU Prime and NTRU key generation as well as Dilithium signing [130]. To perform the attack, they first profile a device with a *known* private key using 90 000 traces while measuring the power consumption. Traces in this context means the number of observed key generation, encapsulation or decapsulation operations. From these traces they build a so-called template [131], which can be used to extract the unknown private key from the target device using a single measurement trace. Furthermore, Ravi et *al.* present a method to recover the Streamlined NTRU Prime private key with a side-channel assisted chosen-ciphertext attack [132]. They demonstrate the capability of a full key recovery with just 3 005 traces for the smallest parameter set and with 4 688 traces for larger parameter sets. For more details on side-channel assisted chosen-ciphertext attacks, see Section 5.3.

To combat these side-channel attacks, dedicated countermeasures aiming at decoupling the connection between secret data and power consumption or EM emanation have been proposed in the past decades. One such technique for that purpose is *masking* which splits secret values into multiple uniform random shares and is described in more detail in Section 5.2.

A second countermeasure is *blinding*, which has seen widespread use in securing implementations of RSA and ECC [133, 134]. Blinding can be applied to the modular exponentiation in RSA and the scalar point multiplication in ECC, by multiplying a random integer b to one of the inputs. Because b is unknown to an attacker, correlating the side-channel information to the inputs is significantly more difficult. There has been some work in blinding PQC lattice implementations, such as in [135], where the polynomial multiplication was blinded in the BLISS signature scheme. Another approach is the blinding in [136] and [137]. Although both papers describe the technique as *masking*, it is more similar to blinding as described in this thesis.

5.2. Masking

Masking is an approach based on Shamir's secret sharing [138–140]. It has been proven as an effective countermeasure against power and EM side-channel attacks by splitting secret values into uniform random shares. An example of the technique is Boolean masking, where a secret value x is split into $d+1$ shares $x^{(i)}$, such that $x = \bigoplus_{i=0}^{d} x^{(i)}$, with $\oplus$ denoting the XOR operation. We call d the masking degree, with for example $d = 1$ implying a *first-order* masked design with two shares. A single share is denoted as $x^{(i)}$ with $0 \leq i \leq d$. A masked variable is denoted as $x^{(0:d)}$. While functions that are linear or affine in the masking domain can be applied to each share individually, we use specialized methods to secure non-linear functions like AND or OR operations. At any occurrence of Boolean operations that involve masked variables, we assume to perform this securely. A special case is the inversion of a secret value (denoted as $\overline{x^{(0:d)}}$), which is performed by inverting one share rather than inverting each share (which would not invert the secret value for odd d). The creation of the initial masked shares requires randomness, as do all non-linear operations.

Another technique is arithmetic masking. Here, the secret value x is again split in $d+1$ shares $x^{(i)}$, but in way such that $x = \sum_{i=0}^{d} x^{(i)}$. This allows operations such as additions and multiplications to be performed share-wise. However, this comes at the cost of not being able to perform Boolean operations.

5.2.1. Masking Security Models

In order to verify and evaluate the resistance against side-channel attacks of such special functions that operate on masked shares, a range of different attacker models have been proposed. In 2003, Ishai, Sahai, and Wagner [141] introduced the d-probing model which

is still frequently used as an appropriate abstraction. This model allows an attacker to set d so-called *probes* on d wires of a circuit or implementation. This enables the attacker to read the values of the probed wires and variables. The process of probing can be seen analogous to analyzing a circuit with an oscilloscope or a logic analyzer. It is obvious that on a unmasked implementation, even a single probe is sufficient to read the secret variable. However, since masking has split our secret variables into $d+1$ shares, reading d values is not sufficient to reconstruct the original secret variable. We thus can call an implementation secure in the d-probing model if the values gathered from the probes are independent from the secret variable.

However, the d-probing model neither includes glitches nor memory transitions or couplings. Glitches are transient combinatorial recombinations, caused by small variation in the timing and propagation behavior of transistors and wires [142, 143]. Memory transitions are caused by memory recombinations, where registers, FFs and RAM have a different leakage behavior depending on what was stored previously [144, 145]. Finally, coupling leakages are caused by routing recombinations, where routing wires that run close by electrically influencing each other [146]. All three of these phenomena can cause additional leakages, and have been used in successful SCAs. To address these issues, the d-probing model has been extended to a *robust d-probing model* incorporating these phenomena [147, 148]. The robust d-probing model uses *extended probes*. In comparison to the probes used in the d-probing model, which only read a single variable, extended probes can read multiple variables according to the specific extension. For example, a glitch-extended probe reads all inputs of a combinatorial circuit, rather than just a single value. As a result, a glitch-extended probe can properly model the occurrence of glitches. A transition-extended probe reads the values of a memory element across two consecutive clock cycles, again allowing memory transitions to be modeled. Finally, a c-couplings extended probe allows the probe to read up to c wires adjacent to the probed wire [147, 148]. We can call an implementation secure in the robust d-probing model if the values gathered from the extended probes are independent of the secret variable. For example, glitch-extended probes can be defended against by adding additional register stages, which prevent the propagation of glitches. Transition-extended probes can be mitigated for example by zeroing a register before use.

The robust d-probing model by itself is not sufficient to evaluate the composability of atomic building blocks called *gadgets*. Composability allows these gadgets to be arbitrarily combined, which allows us to build complex circuits and algorithms out of comparatively simple gadgets. This divide-and-conquer approach can significantly simplify the effort needed to mask an algorithm. However, in the robust d-probing model, two individual gadgets may be separately evaluated to be secure, while their combination may turn out to be insecure. Examples of gadgets are given in Section 5.2.3.

Hence, Barthe et *al.* introduced Non-Interference (NI) as the first composability notion in 2015 [149]. A key concept in NI is *simulatability*: If any set of probes inside a gadget

can be simulated by probes on some of the gadget's inputs, then the security analysis can focus only on those inputs. This enables probe propagation, where any probes inside of a gadget are replaced with simulator probes on the inputs. In turn, if this gadget is then part of a larger composite design, then the probes can again be replaced with new simulator probes on the inputs of the larger composite design. NI states that a gadget with one shared output is d-NI if any set of at most d_1 probes on internal wires and d_2 probes on its output such that $d_1 + d_2 \leq d$ can be simulated with $d_1 + d_2$ probes on each of its input's shares. Informally, this means that one probe on an internal variable or output of a d-NI gadget propagates to one probe on each of the inputs.

Although NI limits the leakage between shared intermediate results, it does not guarantee full probing security of arbitrary composed circuits. Therefore, Barthe et *al.* presented the notion of Strong Non-Interference (SNI) [150] which ensures full composability of gadgets. SNI states that a gadget with one shared output is d-SNI if any set of at most d_1 probes on internal wires and d_2 probes on its output such that $d_1 + d_2 \leq d$ can be simulated with d_1 probes on each of its input's shares. In contrast to NI, only internal probes are propagated, probes on the output are not propagated. SNI enables trivial composition of any number of gadgets. However, there is a significant overhead involved to ensure a gadget complies to SNI, in the form of additional register stages and fresh randomness required.

Eventually, the concept of Probe-Isolating Non-Interference (PINI) [151] was proposed in order to reduce the overhead introduced by SNI gadgets. PINI is defined as: Given a gadget G over d shares, let I be a set of d_1 probes on internal wires of G and let O be a set of d_2 probes on its outputs. Define A as the set of share indices of the shares in O, with $d_2 = |A|$. Choose I and O so that $d_1 + d_2 \leq d$. The gadget G is d-PINI if and only if for all I and O there exists a set of at most d_1 share indices B such that the probes corresponding to I and O can be simulated using only the shares with indices $A \cup B$ of each input sharing. The security model of PINI ensures that all shared gadgets are *efficiently and trivially* composable in the robust d-probing model. For example, PINI allows XOR operations to be performed trivially share-wise with no fresh randomness required. In contrast, XOR operations in SNI require fresh randomness.

5.2.2. Prior Work on Masking Post-Quantum Cryptography

In the context of PQC, research has focused recently on masked PQC implementations in *software*, mostly for Kyber and Saber. For example, [152] and [153] present masked implementations of Kyber, and [154] and [155] present masked implementations of Saber. A recent paper presents a masked implementation of NTRU for embedded software [156]. Most of these works use the ARM Cortex-M4 as a target platform. However, hardened hardware implementations are also of interest. Previous works again focus on Kyber [157–159] and Saber [158, 160]. All of the above implementations use *algorithmic* masking, i.e., the underlying algorithms were analyzed as to which part must be masked in the Boolean domain, and for which parts arithmetic masking is more efficient. When

switching between Boolean masking and arithmetic masking (or vice-versa), a masking conversion must be performed. Due to the many Boolean and arithmetic operations in PQC algorithms, the masking conversion must often be performed several times, and can lead to a significant overhead.

5.2.3. Gate-Level and Gadget-Based Masking

An alternative to algorithmic masking is gate-level masking. The base concept of gate-level masking is to replace the individual default hardware gates with secure versions called *gadgets*. These secure hardware gates are designed to not leak the inputs and outputs via their power consumption or EM emanation. Early versions of these secure gates aimed at ensuring that the power consumption remained constant, regardless of the input or output. An example of this design is Wave Dynamic Differential Logic (WDDL) [161], which is a type of Dual-Rail with Precharge (DRP) logic. An overview of other DRP logic styles can be found in [162]. These logic gates use differential inputs and outputs and have a pre-charge phase which ensures that transistors switch at every clock cycles, even if the inputs do not change. DRP gates can somewhat be seen as a type of Boolean masking with two shares. However, many of the DRP gates were successfully attacked over the years. Some examples are in [162–165]. These attacks were possible due to effects such as unbalanced routing of the differential signals or glitches in the circuit, as these approaches did not follow the security models explained in Section 5.2.1.

A different approach was presented in [166], where, using the concept of PINI and Boolean masking, Cassiers et *al.* proposed Hardware Private Circuits (HPCs). HPC allows to instantiate an arbitrary-order Boolean masked SecAND gate with two clock cycles latency for one input and one clock cycle for the other input. This gadget is called HPC1. Moreover, in the same paper they optimized this gadget for less randomness demand and denoted it as the HPC2 gadget. With these two SecAND gadgets, one can build any arbitrary complex Boolean function, by combining SecAND gadgets together with gates that are linear in the masking domain, such as XOR and INV gates. These linear gates require no randomness or additional registers, due to the HPC gadgets fulfilling PINI, which also ensures the composability. The HPC2 SecAND gadget can be found in Algorithm 8. We will be referring to this technique as *gadget-based masking* in order to better differentiate it from earlier gate-level masking techniques such as WDDL.

Following this, Knichel et *al.* proposed Generic Hardware Private Circuits (GHPCs) to build more complex gadgets that are PINI [167]. Unlike the HPCs gadgets, which can only implement a two input SecAND, GHPC can directly implement any arbitrary non-linear Boolean function with n inputs and m outputs. In the same work, they also propose GHPCLL, which reduces the latency of GHPC, at the cost of additional randomness. One drawback of GHPC and GHPCLL however is that only a first-order masking degree is possible. In a recent work, Knichel and Moradi presented HPC3 achieving a lower latency than HPC by using more fresh randomness [168].

Finally, Knichel and Moradi also presented COMAR, which featured composable gadgets that can reuse randomness [169]. This allows the required randomness for an arbitrary number of gadgets to be the same six bits. However, because the COMAR gadgets are not PINI, this comes with the cost that XOR gates also require the randomness, as well as two register stages. COMAR also only supports first-order security. Table 5.1 gives an overview of currently available secure hardware gadgets.

Table 5.1.: An overview of currently available hardware gadgets. COMAR's randomness is a special case, because it can be reused for an arbitrary number of gadgets.

Name	Masking Degree d	Randomness	Latency	Function
HPC1 [166]	arbitrary	$d(d+1)/2+d$	2	AND
HPC2 [166]	arbitrary	$d(d+1)/2$	2	AND
HPC3 [168]	arbitrary	$d(d+1)$	1	AND
GHPC [167]	first-order	m	2	$\mathsf{F}:\mathbb{F}_2^n \mapsto \mathbb{F}_2^m$
GHPCLL [167]	first-order	$2^n \cdot m$	1	$\mathsf{F}:\mathbb{F}_2^n \mapsto \mathbb{F}_2^m$
COMAR [169]	first-order	6	2	AND,XOR

Algorithm 8 The PINI SecAND HPC2 gadget [166], for masking degree d with $d+1$ shares. Reg denotes a register stage. $\oplus$ is the XOR, and $\otimes$ is the AND operation.

Input shares $a^{(0:d)}$ for $0 \leq i \leq d$, such that $a = \bigoplus_{i=0}^{d} a^{(i)}$
Input shares $b^{(0:d)}$ for $0 \leq i \leq d$, such that $b = \bigoplus_{i=0}^{d} b^{(i)}$
Output shares $c^{(0:d)}$ for $0 \leq i \leq d$, such that $c = \bigoplus_{i=0}^{d} c^{(i)} = a \otimes b$

1: **for** i from 0 to d **do**
2: **for** j from $i+1$ to d **do**
3: $r_{ji} \xleftarrow{\$} \mathbb{F}_2$
4: $r_{ij} \coloneqq r_{ji}$
5: **end for**
6: **end for**
7: **for** i from 0 to d **do**
8: **for** j from 0 to d, $j \neq i$ **do**
9: $u_{ij} \coloneqq \neg a^{(i)} \otimes \mathsf{Reg}[r_{ij}]$
10: $v_{ij} \coloneqq b^{(j)} \oplus r_{ij}$
11: **end for**
12: **end for**
13: **for** i from 0 to d **do**
14: $c_i \coloneqq \mathsf{Reg}[a^{(i)} \otimes \mathsf{Reg}[b^{(i)}]] \oplus \bigoplus_{j=0, j\neq i}^{d} (\mathsf{Reg}[u_{ij}] \oplus \mathsf{Reg}[a^{(i)} \otimes \mathsf{Reg}[v_{ij}]])$
15: **end for**

5.3. Chosen-Ciphertext Side-Channel Attacks on the Fujisaki-Okamoto Transform

Streamlined NTRU Prime, like many lattice-based KEMs, makes use of the Fujisaki-Okamoto transform to convert a OW-CPA PKE into an IND-CCA2 secure KEM. This protects the KEM from classical chosen-ciphertext attacks. However, the FO transform enables a new class of attacks named Chosen-Ciphertext Side-Channel Attacks (CC-SCAs), sometimes also called side-channel assisted chosen ciphertext attacks. These new attacks, presented for example in [122, 125, 170, 171], exploit side-channel leakages of the FO transform to target the re-encryption. To perform the attack, the adversary carefully constructs ciphertexts so that one bit of the decrypted message depends on a single private key coefficient. In the case of Streamlined NTRU Prime, this would mean for example crafting ciphertexts such that that a single bit of r (Line 5 in Algorithm 3) depends on a single coefficient of the private key polynomials f or g^{-1}. Because this message r is then used in the FO-transform as the input for the deterministic re-encryption, an adversary only has to distinguish whether the target bit is 0 or 1, while analyzing the leakages of the re-encryption. As the FO-transform is a fairly complicated operation, an adversary can analyze a large number of leaking intermediates for the attack. In practical attacks, the number of traces for a successful attack are on the order of a few thousands for many different PQC KEMs, including side-channel protected implementations [171].

The ability of CC-SCA to attack side-channel protected implementations via the FO-transform is particularly worrisome. To analyze the effects, Azouaoui et *al.* in [172] modeled attacks on lattice-based KEMs. Notable, they were able to derive a formula to estimate the minimal number of traces needed for a successful side-channel assisted chosen ciphertext attack, under a worst-case assumption where an attacker can exploit *all* leakages of intermediates. Their approximation can also take into account the number of masking shares and the noise level of the hardware platform in order to assess the impact of masking as a countermeasure. In a concrete example, in order to achieve security against an attack with 10^6 traces for standard 32 bit MCUs, a high-order masked implementation is needed with seven shares. Such a high-order masking in turn comes with a significant performance penalty: The bitsliced ARM Cortex-M4 based implementation of seven-share masked Kyber from [173] takes 58 million clock cycles, of which 96 % are due to the masked re-encryption. In comparison, the unmasked Kyber implementation from the pqm4 project takes just 703 thousand clock cycles on a Cortex-M4 [174].

In short, while the FO-transform provides excellent theoretical security, its presence makes side-channel protected implementations significantly more difficult. Solutions to this problem, apart from increasing the masking degree, include statistical checks to discard SCA-relevant chosen ciphertexts [170], employing zero-knowledge proofs [172] or by replacing the FO-transform with a signature to ensure the ciphertext is valid [175]. However, none of these solutions are suitable for all scenarios, and this area remains an active research topic.

5.4. Automated Masking Tool AGEMA

The AGEMA framework was first introduced in [22] and is available for download at [176]. The tool allows the automatic generation of provably-secure masked hardware implementations, based on Binary Decision Diagrams (BDDs) and secure composable gadgets. AGEMA has been successfully used to generate masked hardware implementations of several symmetric ciphers [22, 168, 169]. The process is as follows:

1. To begin, a hardware design is first synthesized to a gate-level netlist using standard ASIC synthesis EDA tools.
2. The netlist is then annotated, to indicate which inputs are public and which are secret i.e., have to be masked.
3. This netlist is then transformed by AGEMA to a secure version for a given security order.

There are a number of different options for this transformation. The simplest transformation is the naive approach, where each gate that processes secret data is replaced with a secure masked version of that gate. Additional register stages are added where necessary to keep the outputs aligned, as masked gates have an increased latency. Alternatively, AGEMA can first transform the netlist into other representation, such as its Algebraic Normal Form (ANF), BDDs or And-Inverter Graph (AIG) form, and then replace the gates with masked gadgets. For this, AGEMA employs libraries such as CUDD [177] and SYLVAN [178]. This transform may lead to a more optimized design.

Another configurable option is the masked gadget to be used. Possible gadgets currently are HPC1, HPC2, HPC3, GHPC, GHPCLL or COMAR. Each gadget offers different maximum masking degree, randomness requirement and latency. In addition, AGEMA can be configured to produce a pipelined design, which can process new inputs at every clock cycle. Alternatively, AGEMA can use clock gating to ensure all input registers keep their values until the design has finished processing. Using a pipelined design significantly increases the throughput, but also significantly increases the required registers.

5.5. Security Analysis of a Side-Channel Protected Implementation

In order to analyze the concrete security of a side-channel protected implementation, one method is to apply Welch's non-specific (fixed vs. random) t-test [179,180], which is part of the so-called Test Vector Leakage Assessment (TVLA) [181]. Welch's t-test gives us the probability that the mean μ of two data sets are different and can be used to detect if an implementation leaks information via side channels. To perform the t-test, we first measure the power consumption or EM emanation of a design with an oscilloscope for two different setups: For the first setup A, we use a certain key k, while performing the

cryptographic operation (e.g., decapsulation) on a fixed input. For the second setup B, we again use key k, but use a random input. We then compute the t-test statistic for every sampling point:

$$t = \frac{\mu_A - \mu_B}{\sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}}} \tag{5.1}$$

where μ_A, μ_B are the respective sample means, s_A^2, s_B^2 the respective sample variance and n_A, n_B the number of traces in each group. Based on the magnitude of t at all sampling points, we can estimate whether a design leaks information or not, assuming we have gathered enough traces. In [180], they recommend to capture several million traces for a side-channel protected design and to use a threshold of ± 4.5. Equation 5.1 is a *first-order* t-test and is sufficient to evaluate a first-order masked implementation. To evaluate higher-order masked implementations, we require a higher order t-test: a masked implementation of order d requires a d^{th} order t-test. To do so, we replace the mean μ and variance s^2 with higher order statistical moments. The required changes are given in Equations 5.2 through 5.6, where M_i is the i^{th} raw moment, SM_i is the i^{th} standardized moment and CM_i is the i^{th} central moment [180].

$$\text{(1st order)} \quad \mu = M_1 \qquad s^2 = CM_2 \tag{5.2}$$

$$\text{(2nd order)} \quad \mu = CM_2 \qquad s^2 = CM_4 - CM_2^2 \tag{5.3}$$

$$\text{(3rd order)} \quad \mu = SM_3 \qquad s^2 = \frac{CM_6 - CM_3^2}{CM_2^3} \tag{5.4}$$

$$\text{(4th order)} \quad \mu = SM_4 \qquad s^2 = \frac{CM_8 - CM_4^2}{CM_2^4} \tag{5.5}$$

$$\text{(ith order)} \quad \mu = SM_i \qquad s^2 = \frac{CM_{2i} - CM_i^2}{CM_2^i} \tag{5.6}$$

However, it is important to note that TVLA and the t-test are not foolproof. Although they give us an indication on the side-channel security of a design, both false-positive and false-negatives are possible. For example, the side-channel protected implementations that were broken with a CC-SCA attack in Section 5.3 all passed their respective t-tests. An in-depth analysis of the limitations of the t-test can be found in [182].

An additional, more rigorous method to evaluate the side-channel protection is to perform formal verification. For hardware implementations, this can be performed using the verification tool VERICA [183]. VERICA is constructed based on the verification concepts developed in the side-channel analysis tool SILVER [184] and the fault-injection analysis tool FIVER [185]. The formal verification of a target design is performed based on its (Verilog) gate-level netlist which is transformed into a Direct Acyclic Graph (DAG) serving as circuit model. Each node in the DAG is associated with a BDD representing

the Boolean function of the corresponding gate. This data structure allows efficient applications of statistical checks verifying side-channel security in the glitch-extended probing model and composability notions.

Part II.

High-Speed and Low-Area Implementations

6. Implementation

This chapter will present the basic functionality and architecture of all core hardware modules that are needed for a Streamlined NTRU Prime hardware implementation. In addition, multiple modules are presented in different variations, as they were optimized over the course of this thesis. There are two main optimization targets: high-speed implementations, intended for heavy-duty applications such as servers, and lightweight low-area implementations intended for embedded devices. The implementations presented in this chapter are based on my published implementations [10] and [17], with some additional improvements and extensions. Side-channel protected designs will not be covered in the chapter. They are presented separately in Chapter 8.

Throughout this chapter, we will generally be referring to an FPGA as the target platform, as this was the platform used for development and testing. However, all designs can also be synthesized for an ASIC. In these cases, FPGA specific parts such as BRAM or distributed RAM are synthesized to their ASIC counterparts (for example Static Random Access Memory (SRAM)).

6.1. Karatsuba & Schoolbook Multiplication

In Streamlined NTRU Prime, all multiplications are with one polynomial in $\mathcal{R}/3$, and the second either also in $\mathcal{R}/3$ or in $\mathcal{R}/q$. Multiplication where both polynomials are in $\mathcal{R}/q$ do not normally occur (the only exception here is batch inversion using Montgomery's trick, see Section 4.2). The "big×small" structure of the multiplication was previously explored for other PQC KEMs in e.g., [93] and [88], and allows a number of optimizations: While the $\mathcal{R}/q$ polynomial has signed 13 bit coefficients, the small $\mathcal{R}/3$ polynomial only has signed 2 bit coefficients. To be specific, these 2 bit coefficients can only have the values -1, 0 and 1. This results in the coefficient multiplier being almost trivial to implement when using the schoolbook technique because the multiplication of a 13 bit number with either -1, 0 and 1 is exceptionally lightweight in hardware. Using a pure product scanning schoolbook multiplication [92] is thus very simple: The polynomials are stored in BRAM and one coefficient from each polynomial is retrieved each clock cycle, multiplied together, and then accumulated onto the result polynomial coefficient, which is then written back to BRAM. However, such a serial computation would take a long time to complete, in the order of $\mathcal{O}(p^2)$ clock cycles.

In order to improve performance, we use a combination of Karatsuba multiplication and product scanning schoolbook multiplication (recall Section 4.3). We apply a single layer of Karatsuba multiplication. This splits a multiplication of degree p into three multiplications of degree $\lceil p/2 \rceil$. The three partial multiplications are then done in parallel with product scanning schoolbook multiplication. During the schoolbook multiplication,

we also make use of the dual port BRAM to read and write two words per clock cycle, doubling the speed of the multiplication. The use of Karatsuba does make the partial multiplication slightly more complicated, as one of the partial multiplications involves the addition of the lower and upper part of the small $\mathcal{R}/3$ polynomial. This leads to a polynomial in $\mathcal{R}/q$ with 3 bit coefficients, with values between -2 and $+2$.

We first presented this polynomial multiplier in [10], as part of a low-area design. As part of a study to investigate further improvements, we adapted the multiplier to perform two layers of Karatsuba, for a total of nine partial multiplications. All of the partial multiplication are again performed in parallel. Due to the additional recursion layer, the polynomial that was originally small now has coefficients between -4 and 4, which increases the complexity of the coefficient multiplier further. As a result, although the additional layer brought a further speed increase, the increased memory and area usage is no longer worth the speedup. Details on the speed and area usage can be found in Chapter 7, and the code in [12].

6.2. Parallel $\mathcal{R}/q$ Schoolbook Multiplier

The schoolbook multiplier presented in Section 6.1 operates in a serial manner, computing one coefficient at a time. Instead of using a technique like Karatsuba to accelerate the multiplication, we can also parallelize the computation, effectively unrolling the inner loop in Line 3 in Algorithm 5. This type of schoolbook multiplication has been previously presented for other PQC algorithms such as in [88, 93, 186]. It consists of an Linear Feedback Shift Register (LFSR), an accumulator register, and a large number of multiply accumulate units. Two different implementations, based on the same overall design architecture, are presented in this thesis: A high-speed, high-area implementation and a much smaller, but also slower implementation. Both are similar with regards to the speed-area product. They also have very simple memory access patterns. The differences between the two is that the faster implementation stores all values in flip-flops, whereas the compact implementation uses distributed RAM. The architecture is shown in Figure 6.1.

The high performance and efficiency of this design is based on the fact that as mentioned in Section 6.1, the multiplications in Streamlined NTRU Prime by default always have at least one $\mathcal{R}/3$ polynomial as input, i.e., a "big×small" multiplication. This feature allows the individual Multiply-Accumulate (MAC) units to be very simple, as only a small number of bit operations are needed. In turn, this causes the MAC units to have a very small footprint in an FPGA or ASIC, allowing us to instantiate a large number of them. In addition, we do not include any modular reduction, again to simplify the MAC units. Instead, the accumulator register is of sufficient size such that no overflow can occur, and we reduce at the end of the multiplication. The calculation of the minimum register size is done in the same way as for the NTT and CRT in Section 4.4.2: The coefficients after a $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplication will be in range $[-\frac{p(q-1)}{2}, \frac{p(q-1)}{2}]$, which is

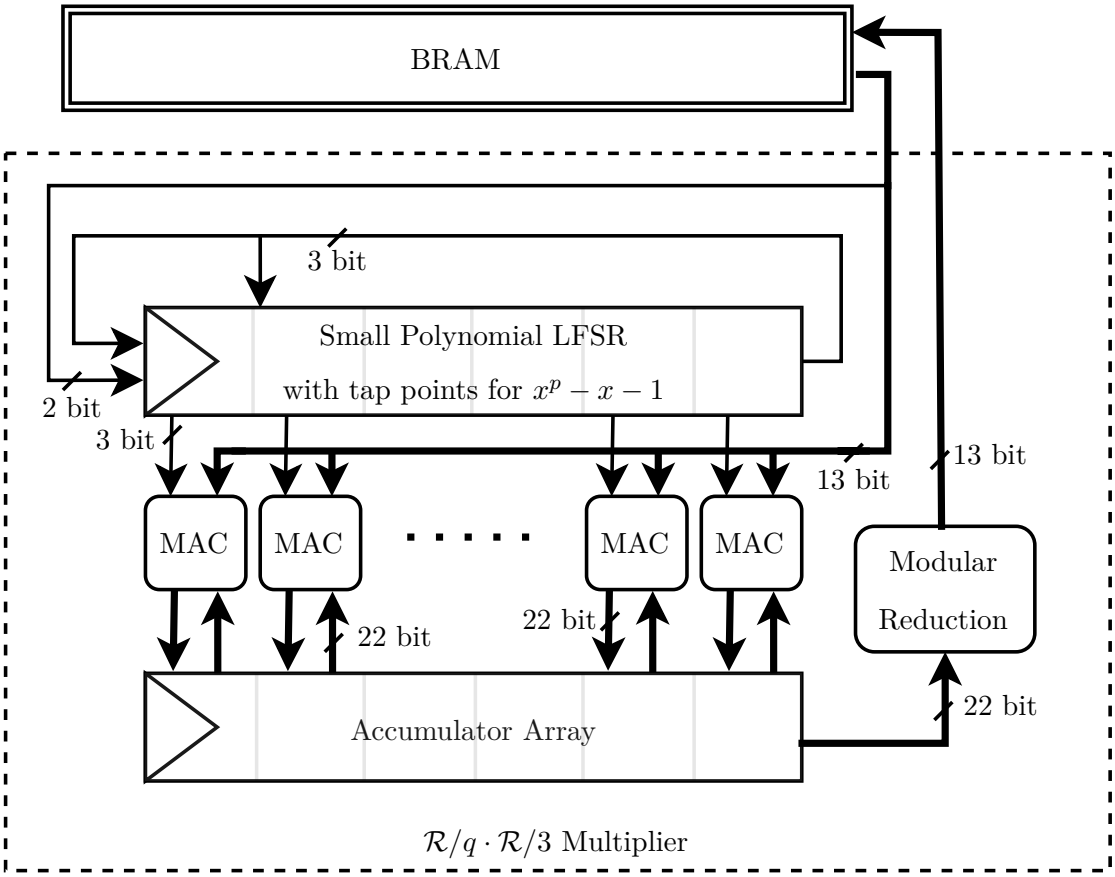


Figure 6.1.: Architecture of the $\mathcal{R}/q$ parallel schoolbook polynomial multiplier for the parameter set sntrup761 [17]. The accumulator array has a size of $p \cdot 22$ bits. The blocks with the label MAC are described in Algorithm 9. The difference between the high-speed and the low-area multiplier are in the number of MAC units, and whether the accumulator array and small polynomial LFSR are implemented in flip-flops or in distributed RAM.

a $\lceil \log_2(q \cdot p) \rceil$ bit signed number. The algorithmic description of the MAC unit can be found in Algorithm 9.

Before the multiplication starts, the small $\mathcal{R}/3$ polynomial is loaded into an LFSR of length p, with the tap points set to correspond to the polynomial of the Streamlined NTRU Prime ring, $\mathcal{R}/q = \mathbb{Z}/q[x]/(x^p - x - 1)$. Three bits are needed per coefficient because the shifts and tap points can lead to coefficients in the range from -2 to 2 [9].

Proof. Write the small $\mathcal{R}/3$ polynomial g as $\sum_{j=0}^{p-1} g_j x^j$. By assumption $g_0, g_1, \ldots, g_{p-1} \in \{-1, 0, 1\}$. We can observe that $xg \bmod x^p - x - 1 = g_{p-1} + (g_0 + g_{p-1})x + g_1 x^2 + \cdots + g_{p-2}x^{p-1}$, where $g_0 + g_{p-1} \in \{-2, -1, 0, 1, 2\}$ while all other coefficients are in $\{-1, 0, 1\}$. More generally, for $0 \leq i < p$, we get $x^i g \bmod x^p - x - 1 = g_{p-i} + (g_{p-i} + g_{p-i-1})x +$

Algorithm 9 Single coefficient Multiply-Accumulate (MAC) algorithm. Note that no modulo calculation is performed here.

Param p, q: Parameters from Streamlined NTRU Prime
Input a: a $\lceil \log_2(q \cdot p) \rceil$ bit signed number
Input b: a $\lceil \log_2(q) \rceil$ bit signed number
Input c: a 3 bit signed number with $-2 \geq c \geq 2$
Output The $\lceil \log_2(q \cdot p) \rceil$ bit signed result $a + b \cdot c$

1: $r_{-2} \coloneqq -b \ll 1$
2: $r_{-1} \coloneqq -b$
3: $r_0 \coloneqq 0$
4: $r_1 \coloneqq b$
5: $r_2 \coloneqq b \ll 1$
6: **return** $a + r_c$

$\cdots + (g_{p-2} + g_{p-1})x^{i-1} + (g_{p-1} + g_0)x^i + g_1 x^{i+1} + \cdots + g_{p-i-1}x^{p-1}$ with all coefficients in $\{-2, -1, 0, 1, 2\}$. □

Once the $\mathcal{R}/3$ polynomial is fully shifted into the LFSR, the multiplication can begin. During multiplication, one coefficient from the $\mathcal{R}/q$ polynomial is retrieved from the BRAM at a time. This coefficient is then multiplied in parallel with every single coefficient in the LFSR and added coefficient-wise to an accumulator register. This is effectively an unrolled version of the inner for loop in Line 3 in Algorithm 5. However, no modulo calculation is performed at this step. The LFSR is then shifted once (Line 6 in Algorithm 5), and the next coefficient from the $\mathcal{R}/q$ polynomial is retrieved. We shift the small $\mathcal{R}/3$ polynomial as this requires fewer hardware resources. This repeats for every coefficient from the $\mathcal{R}/q$ polynomial (Line 2 in Algorithm 5). After this, the accumulator register contains the completed polynomial multiplication. The register contents are then sent to the multiplier output, where they are reduced modulo q. Because of the LFSR, no additional polynomial reduction is required.

For the high-speed schoolbook multiplier, p MAC units are instantiated. As a result, the inner for loop is fully unrolled and one coefficient from the $\mathcal{R}/q$ polynomial can be processed per clock cycle. For the low-area implementation using distributed RAM, the loop is only partially unrolled, and 24 MAC units are instantiated for the sntrup761 parameter set. This number comes from the value of p, and the size of the smallest distributed RAM blocks. In Xilinx FPGA's, the LUT can be configured as 32 bit dual-port RAM, with one read/write port, and one read-only port. With $p = 761$, and $\lceil 761/24 \rceil = 32$, it means that 24 MAC units pack the RAM as densely as possible. Thus, every 32 clock cycles a new coefficient from the $\mathcal{R}/q$ polynomial is processed, and the multiplier thus also takes 32 times as many cycles.

By default, it takes p clock cycles to shift the $\mathcal{R}/3$ polynomial into the LFSR. It also takes p clock cycles to shift the result out of the accumulator array, during which the

Algorithm 10 Single coefficient MAC algorithm for the $\mathcal{R}/3$ multiplier.

Input a: a 2 bit signed number with $-1 \geq a \geq 1$
Input b: a 2 bit signed number with $-1 \geq b \geq 1$
Input c: a 2 bit signed number with $-1 \geq c \geq 1$
Output The 2 bit result $a + b \cdot c \in \mathbb{Z}/3$

1: $r_{-1} \coloneqq -b$
2: $r_0 \coloneqq 0$
3: $r_1 \coloneqq b$
4: **return** $a + r_c \bmod^{\pm} 3$

accumulator array is also set to zero. Both of these operations can be interleaved to save time, i.e., a new $\mathcal{R}/3$ polynomial can be shifted into the LFSR while the accumulator array is shifted out. In addition, as an extension of [17], the multiplier is adapted to load two coefficients of the $\mathcal{R}/3$ polynomial into the LFSR per clock cycle. This can be configured at run-time. In a similar fashion, the multiplier can now output two words of the accumulator array per clock cycle. This reduces the setup time needed before and after a multiplication, though it does require a second modulo reduction unit.

It should be noted that this form of massively parallel schoolbook multiplication cannot be combined with the Karatsuba multiplication. This is due to the fact that the LFSR already performs the $\mathcal{R}/q$ reduction automatically. However, in Karatsuba multiplication, the polynomial reduction has to be performed after all partial multiplications have been reassembled. In contrast, the parallel schoolbook multiplier would perform the polynomial reduction also on the partial product, leading to incorrect results.

6.3. Parallel $\mathcal{R}/3$ Schoolbook Multiplier

A further extension of the work from [17] is a dedicated $\mathcal{R}/3 \cdot \mathcal{R}/3$ multiplier. A $\mathcal{R}/3$ multiplication is performed during the decapsulation of Streamlined NTRU Prime (Line 5 in Algorithm 3). In [17], we use the $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplier for this task, with a subsequent coefficient reduction back to $\mathcal{R}/3$. However, a dedicated $\mathcal{R}/3$ multiplier allows a further speed improvement in the decapsulation. Due to the area overhead, this is only really viable for a high-speed design. The architecture is nearly identical to the $\mathcal{R}/q$ multiplier and can be found in Figure 6.2. The main difference are the MAC units: They are even simpler because all coefficients, inputs and outputs are in $\mathbb{Z}/3$ i.e., in the range of -1 to 1. This means there are only a total of 27 possible input combinations and three possible outputs. The MAC units for the $\mathcal{R}/3$ multiplier are described in Algorithm 10. In addition, there is no option to use distributed RAM, flip-flops are always used, nor is their an option to reduce the number of MAC units: We always instantiate p MAC units. A further optimization is the integration of the weight check (Line 6 in Algorithm 3) directly into the $\mathcal{R}/3$ multiplier. We describe this in more detail in Section 6.9.

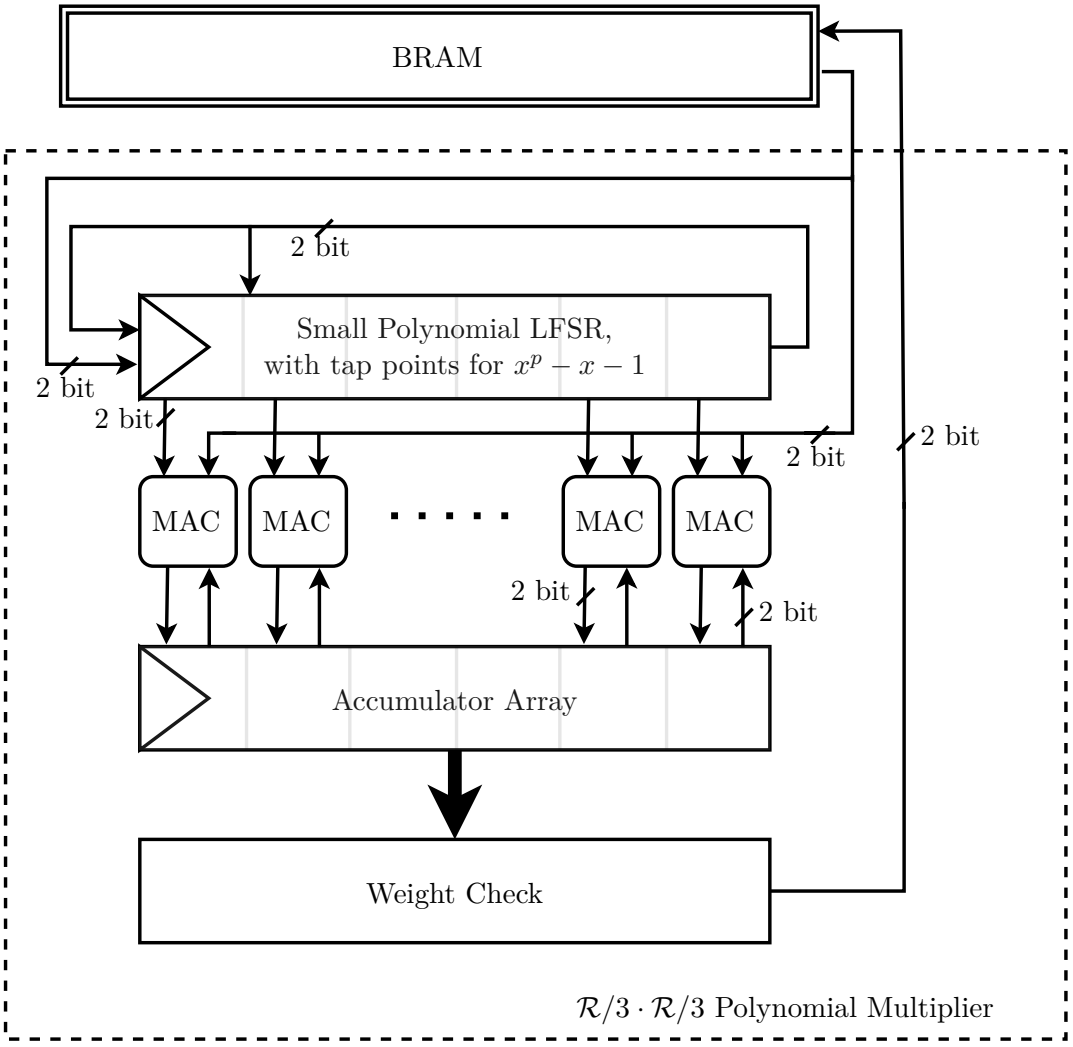
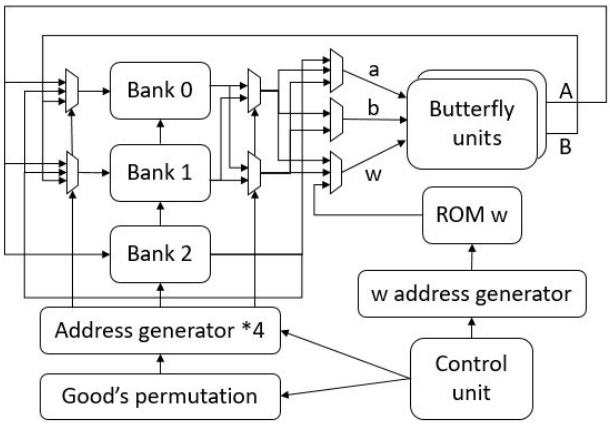


Figure 6.2.: Architecture of the $\mathcal{R}/3$ parallel schoolbook polynomial multiplier.

6.4. Architecture of $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT Multiplier

In Montgomery's trick for batch inversion (see Section 4.2), a multiplication of two polynomials in $\mathcal{R}/q$ is performed. As a result, we cannot use the schoolbook multiplier presented in Section 6.2. Instead, we use an NTT based multiplication employing Good's trick and the CRT (see Section 4.4). This multiplier was first presented in [17] and was initially developed by my co-authors from the National Taiwan University and the Academia Sinica. However, the integration into the larger design as well as some optimizations was done by me. The architecture of the NTT multiplier is shown in Figure 6.3 and is originally based on the NTT/INTT architecture from [104]. This NTT multiplier can be used for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication for the parameter sets sntrup653 and sntrup761.

Coefficients in polynomials $f(x)$ and $g(x)$ are partitioned into those of $f_{y^0}(z)$, $f_{y^1}(z)$, $f_{y^2}(z)$, $g_{y^0}(z)$, $g_{y^1}(z)$, and $g_{y^2}(z)$, as described in Section 4.4.1. We employ three memory banks labeled 0, 1 and 2. Each z-polynomial is first put into banks 0 and 1. With a careful design of the four address generators to the reading and writing channels of bank 0 and 1, the z-polynomials are passed through the 3 Butterfly units (for $\mathbb{Z}/7681$, $\mathbb{Z}/12289$, and $\mathbb{Z}/15361$ respectively) and the corresponding NTT vectors are calculated. Bank 2 then stores the result of the summation of the point multiplications. The content in bank 2 contains the NTT vectors of $h_{y^0}(z)$, $h_{y^1}(z)$, and $h_{y^2}(z)$. The 3 INTT operations are then performed on the NTT vectors, in order to compute $h(x) = f(x)g(x)$. At this point, each coefficient is a 3-tuple with each entry representing the coefficient modulo 7681, 12289, 15361, respectively. The CRT operation is then done to find each coefficient modulo 4591, and then the reduction of $x^p - x - 1$ is computed, after which we receive the final $h^{(r)}(x) = h(x) \bmod x^p - x - 1$.



Address	Bank 0	Bank 1	Bank 2
0~255	f y^0	f y^0	y^0
256~511	f y^1	f y^1	
512~767	f y^2	f y^2	y^1
768~1023	g y^0	g y^0	
1024~1279	g y^1	g y^1	y^2
1280~1535	g y^2	g y^2	

Figure 6.3.: Architecture of Good's trick NTT multiplication [17].

This multiplier is used for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication during batch inversion (see Section 6.6) and takes 35 463 clock cycles. The control unit consists of the following stages: *load*, *NTT*, *point_mul*, *reload*, *INTT*, *crt*, and *reduce* stages. When the product of the polynomials is ready, the control unit falls into the *finish* stage, and the result can be fetched out of the multiplier.

- In the *load* stage, the coefficients of the to-be-multiplied polynomials f and g are stored in bank 0 and bank 1. The address is determined using Good's permutation and the address generators. 3 090 clock cycles are consumed in this stage.
- In the *NTT* stage, read one coefficient from bank 0 and one coefficient from bank 1, pass them to the butterfly unit, and write the output A and B back to the same address of bank 0 and bank 1. There are six NTT operations with 512 points to be done, and each NTT operation needs 2 315 cycles ($256 \cdot 9 + \epsilon$). It takes 13 890 cycles in this stage.

- In the *point_mul* stage, we read out two coefficients from the same bank and multiply them. Then we decide which address to write to bank 2 according to the power of y modulo $y^3 - 1$. We write the result into bank 2 directly for the first set of point multiplications. For the second and third sets of point multiplications, we read the value from bank 2 in advance, add it into the result of the point-multiplication, and save it back to bank 2. With this design, we complete the addition of the 3 sets of point multiplication which influence the value of $h(x)$ in this stage. There are $3 \cdot 3$ sets of 512-point multiplications to be calculated, and it takes 4 616 $(512 \cdot 9 + \epsilon)$ cycles.

- In the *reload* stage, read coefficients from bank 2 and write them back to bank 0 and bank 1 after the nine point multiplications. 1 541 cycles are consumed here.

- In the *INTT* stage, the process is similar as in the *NTT* stage. The differences are that the butterfly units and the ω address generator are now operating in the inverse mode. There are 3 INTT operations to be done, and 6 945 cycles are consumed in this stage.

- In the *crt* stage, the DSP slices in the three butterfly units are used to calculate the partial result $h'_{i,a}q_bq_c$ using the CRT. All of the partial results are then summed up as one integer, which is the input of the modulo q unit. After this $f(x)g(x)$ is ready but without modulo $x^p - x - 1$ applied. It takes 3 080 cycles to complete this stage.

- In the *reduce* stage, $h^{(r)}(x) = f(x)g(x) \bmod x^p - x - 1$ is evaluated. A register h_d is used to help the latter addition. The coefficient of x^i in $f(x)g(x)$ for $0 \leq i \leq 760$ (denoted as h_i) are copied sequentially into bank 2, and afterwards we set $h_i \triangleq 0$ during the duplication of the lower coefficients. h_i for $761 \leq i \leq 1520$ is then sequentially loaded, and h_{i-761} is loaded from bank 2 simultaneously. $h^{(r)}_{i-761} \triangleq (h_i + h_d + h_{i-761}) \bmod q$ is computed and saved into bank 2 at the next cycle, and we set $h_d \triangleq h_i$ simultaneously for the next cycle. After h_{1520} is processed, h_{760} is loaded, and $h^{(r)}_{760} \triangleq (h_d + h_{760}) \bmod q$ is computed and saved into bank 2. This stage takes 1 530 cycles.

- $h^{(r)}(x)$ is ready in bank 2 at the *finish* stage. To reduce the critical paths while fetching data from bank 2, a pipelined approach in the address assignment is applied, which results in some overhead. Fetching $h^{(r)}(x)$ takes 770 cycles.

We inspect how the coefficients in the polynomial $f_{y^i}(z)$ and $g_{y^i}(z)$ are stored in the memory banks. One z-polynomial requires 512 cells as the storage of coefficients, and we save half of the coefficients in 256 cells of bank 0 and the other half in 256 cells of bank 1. This design allows us to feed the inputs simultaneously into the butterfly units, and we use the efficient in-place memory addressing introduced in [187], which provides the formula of bank index $B(\cdot)$ and the lower bits of the address $A_l(\cdot)$. The higher bits of

the address $A_h(\cdot)$ just indicate which z-polynomial is being addressed. The bank index and address are given by

$$B(z^i, h_{y^j}(z)) = i[8] \oplus i[7] \oplus ... \oplus i[0] \tag{6.1}$$

$$A_l(z^i, h_{y^j}(z)) = i[8:1] \tag{6.2}$$

$$A_h(z^i, h_{y^j}(z)) = \begin{cases} j, & h = f \\ j+3, & h = g \end{cases} \tag{6.3}$$

$$A(z^i, h_{y^j}(z)) = 2^8 A_h(.) + A_l(.) \tag{6.4}$$

It should be noted that in the *reload* stage and at the end of multiplication, as the NTT itself re-arranges the order of the coefficients such that the address in one polynomial is *bit reversed*, the lower 9 bits of the address need to be reversed. The higher 3 bits do not join the bit reversal.

6.5. Generation of Short Polynomials and Fixed-Weight Sampling

During the encapsulation and key generation in Streamlined NTRU Prime, a short polynomial has to be created. A short polynomial has a fixed weight of exactly w coefficients that are either 1 or -1, with all other coefficients being zero. For this, the original Streamlined NTRU Prime paper suggests using a sorting network [7] and using a sorting algorithm is a well-established method to randomly shuffle a list in constant-time [88, 188]. In our case, a list of p 32 bit random numbers is created. For the first w, the least significant bit is set to 0 so that the number is always even. For the others, the lowest two bits are set to $(0, 1)$. This list of numbers is then sorted, after which the upper 30 bit are discarded. The remaining 2 bit numbers are then subtracted by one. As a result, exactly w elements are either 1 or -1, and the rest are all zero. An alternative method for generating short polynomials would be a shuffling algorithm such as Fisher–Yates, as used by the Dilithium hardware implementation in [189]. However, in Dilithium, a *public* polynomial is sorted, whereas in Streamlined NTRU Prime, a *secret* polynomial is sorted, and thus requires a constant-time algorithm. Because the Fisher–Yates shuffle is difficult to implement in constant-time [88, 188], we do not consider it an option.

The reference C implementation of Streamlined NTRU Prime [9], as well as our hardware implementation in [10] use a constant-time sorting network, originally presented in [7]. The C-code of the sorting algorithm can be found in Listing 6.1. However in hardware, we can use a faster method in the form of the radix sorting algorithm [190]. Radix sort is an extremely fast sorting algorithm, offering $\mathcal{O}(n)$ speed compared to the $\mathcal{O}(n \log n)$ of the sorting network used in [10] and [9]. The use of radix sort for Streamlined NTRU Prime was first presented in our implementation in [17]. However, radix sort has the drawback of having input dependent addressing, which would disqualify it for memory architectures that have a cache due to timing side-channel leakages. Because

```
void minmax(uint32 *x,uint32 *y) {
  uint32 xi = *x; uint32 yi = *y;
  uint32 xy = xi ^ yi;
  uint32 c = yi - xi;
  c ^= xy & (c ^ yi ^ 0x80000000);
  c >>= 31;
  c = -c;
  c &= xy;
  *x = xi ^ c; *y = yi ^ c;
}
void uint32_sort(uint32 *x,int n) {
  int top,p,q,i;
  top = 1;
  while (top < n - top) top += top;
  for (p = top;p > 0;p >>= 1) {
    for (i = 0;i < n - p;++i)
      minmax(x + i,x + i + p);
    for (q = top;q > p;q >>= 1)
      for (i = 0;i < n - q;++i)
        minmax(x + i + p,x + i + q);
  }
}
```

Listing 6.1: The C code of the constant-time sorting network. The minmax function compares and swaps the two inputs [7, 9].

the BRAMs on an FPGA (or the SRAM in an ASIC) do not have any sort of cache, we can safely implement the algorithm. Our implementation is based on the radix sorting algorithm in the SUPERCOP benchmark suite [191]. A comparison of different sorting algorithms is in Table 7.1.

A further optimization we have implemented is the pregeneration of short polynomials. Because short polynomials can be generated independently of the operation (encapsulation or key generation) or any other input (e.g., the public key), we can pregenerate a short polynomial, instead of generating it on-demand. This pregenerated short polynomial is then cached and is output when the encapsulation or key generation starts. Once it has been output, we can use the rest of the time spent on encapsulation or key generation to pregenerate a new short polynomial for the next operation. This in particular speeds up encapsulation because the rest of the modules do not have to wait until the sorting has completed. Note that this pregeneration is also possible for NTRU, but not Kyber, and would allow for a similar speed-up.

The one case where this pregeneration would not be possible is if an encapsulation starts immediately after power on. In that case, the encapsulation would have to wait until a short polynomial is generated. Further encapsulations however would be able to use a cached pregenerated short polynomial, so only the very first encapsulation would

be delayed. However, the described scenario is unlikely to occur in the real word because it disregards aspects such as the loading of the public key (from e.g., a flash storage), which will likely take longer than the sorting.

A further optimization in the generation of short polynomials for very high-speed designs is the usage of *multiple* sorting modules: If the speed of sorting is not sufficient, then multiple sorting modules can operate independently in parallel. Each individual module pregenerates and caches a short polynomial, and outputs it when requested in a round-robin manner, before starting over. This allows short polynomials, averaged over time, to be generated at a much higher rate.

6.6. Batch Inversion using Montgomery's Trick

To accelerate the inversion during key generation, we employ batch inversion using Montgomery's trick. For the polynomial inversion itself, we use the constant-time extended GCD algorithm from [87] (see Section 4.1 for more details). This algorithm uses a constant number of *division steps* (see Equation 4.1) to calculate the inverse of the input polynomial. This algorithm is used by the reference implementation of Streamlined NTRU Prime [9] and is also used in the previously published hardware implementations [10, 17] as part of this thesis. The modular multiplication and subtractions are implemented to employ the DSP slices of FPGAs. The division by x in Equation 4.1 is implemented by shifting the coefficients in memory. In addition, we implement the swap condition in Equation 4.1 with the help of multiplexors that toggle which memory bank is used for the divstep.

We also modify the extended GCD algorithm to enable a vectorized computation of the modular multiplication within the divstep, effectively unrolling the loops in Line 28 and Line 29 in Listing A.1. Increasing the unroll factor proportionally decreases the total number of cycles for an inversion. The architecture of the $\mathcal{R}/q$ inversion can be found in Figure 6.4. We do not consider alternative inversion methods, such as Fermat's method or Hensel lifting, because they are either slower, not constant-time or are not applicable to the rings used in Streamlined NTRU Prime [87, 90]. In addition, we also improve on our design from [17] by adding additional pipeline stages, in order to shorten the critical path and increase the maximum clock frequency.

The use of batch inversion for Streamlined NTRU Prime in a hardware design was first presented in [17], as part of this thesis. We only implement batch inversion for the inversion in $\mathcal{R}/q$. For inversion in $\mathcal{R}/3$, it is more efficient to simply increase the unroll factor, as the modular multiplication operation in $\mathcal{R}/3$ is trivial (see also Algorithm 10). For example, with an unroll factor of 32 the inversion in $\mathcal{R}/3$ takes 47 166 cycles. The inversion in $\mathcal{R}/3$ also has the potential of having non-invertable polynomials. We skip the invertibility check and perform rejection-sampling instead: In case of a non-invertible $\mathcal{R}/3$ polynomial, we simple redo the inversion with a new $\mathcal{R}/3$ polynomial. However for

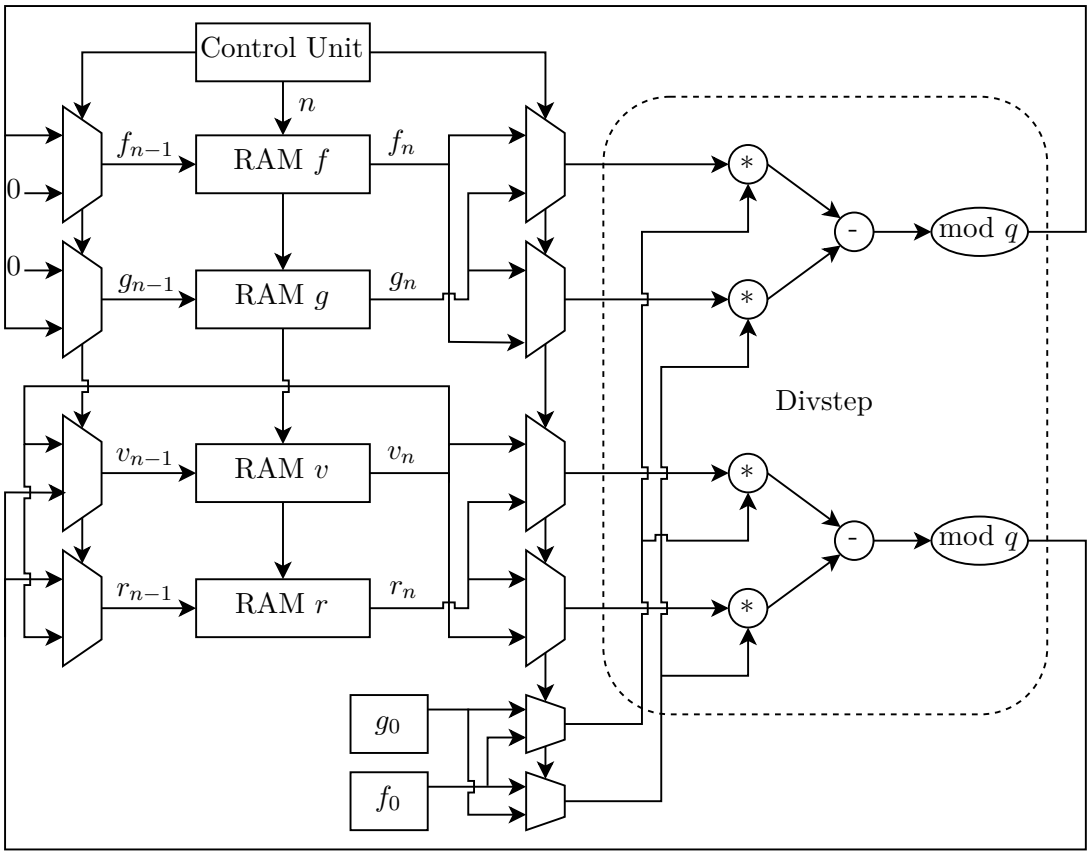


Figure 6.4.: The architecture of the $\mathcal{R}/q$ inversion module using the extended GCD algorithm [17]. The to-be-inverted polynomial is loaded into RAM g. At the start of the algorithm, RAM v stores an all-zero polynomial, RAM r the polynomial $(3^{-1} \bmod q, 0, ..., 0)$ and RAM f the polynomial $(1, 0, ..., 0, -1, -1)$ (f is thus guaranteed to be coprime with g). The final result is stored in RAM v. The section marked "Divstep" is the part that is replicated when loop unrolling is applied. This also requires wider read/write ports to the RAM. The architecture of the $\mathcal{R}/3$ inversion is identical, other that all arithmetic operations are performed in $\mathcal{R}/3$.

batch inversion, we would have to check every polynomial for invertibility, as a single non-invertible polynomial would force us to redo the entire batch. As short polynomials in $\mathcal{R}/q$ are guaranteed to be invertible, this issue does not occur there.

Doing batch inversion has an additional caveat: For a batch size of n, it requires n multiplications where both polynomials are in $\mathcal{R}/q$ (Line 7 in Algorithm 4). This is an issue, as the polynomial multiplier for Streamlined NTRU Prime normally always has one operand in $\mathcal{R}/3$. This means we cannot use our schoolbook multiplier because the multiplier has optimizations that rely on one operand being in $\mathcal{R}/3$ (see Section 6.2). As a result, we add a second multiplier to our design, namely the NTT multiplier with a CRT map for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication (see Section 6.4).

Due to the additional $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier, batch inversion is not automatically the optimal way of inverting polynomials in $\mathcal{R}/q$. This is because the additional multiplier consumes hardware resources that could otherwise be used to implement a higher unroll factor for the $\mathcal{R}/q$ inversion. In addition, larger batch sizes require more BRAM to store intermediate results. Depending on the speed and hardware consumption of non-batch inversion, batch inversion and multiplication respectively, together with the available hardware resources and batch size, the optimal solution varies. A contour plot that shows the minimum batch size needed for Montgomery's trick to be worthwhile for different inversion and multiplication speeds is shown in Figure 6.5. As an example, assume the three multiplications take 40 000 cycles in total, which is roughly how long two $\mathcal{R}/q \cdot \mathcal{R}/3$ and one $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication take in our design for the parameter set sntrup761. At the same time, assume that with the extra hardware resources, we could alternatively accelerate the inversion by a factor of 2. According to the plot, a batch size of 4 would be sufficient for Montgomery's trick to be worthwhile. In practice, we recommend using batch sizes of 5, 21 and 42 for the parameter set sntrup761. These sizes are found via experimentation and pack the 36kbit BRAM available in Xilinx FPGAs as densely as possible. Table 7.14 lists the additional BRAM cost for the different batch sizes, as well as the associated cycles. For FPGAs and ASICs with memories of different sizes, the optimal batch size would vary accordingly.

The additional memory during batch inversion is needed to store the intermediate polynomials a_i, a_i^{-1}, f_i and f_i^{-1} for $0 \leq i \leq n$ (see Algorithm 4), as well the n g_i and g_i^{-1} polynomials. Relatively little memory is needed to store the f_i polynomials because these are short polynomials in $\mathcal{R}/3$, and only require two bit per coefficient. The same applies to g_i and g_i^{-1}, as they are small polynomials in $\mathcal{R}/3$. Our design in [17] stores all polynomial arrays in their entirety and begins the computation of the public key polynomials h_i once the batch inversion is complete. However, this is not actually necessary, and can be optimized to reduce the memory footprint: In Line 7 of Algorithm 4, f_i^{-1} is computed. We can then immediately use f_i^{-1} to compute the corresponding public key polynomial h_i (Line 6 in Algorithm 1), output the private key and public key, and then discard f_i^{-1}. In a similar fashion, once a_i^{-1} has been used to compute f_i^{-1} (Line 8 in of Algorithm 4), it is also not needed further and can be discarded. This allows us to reduce the memory cost because we only ever have to use enough memory to store a single polynomial of each f_i^{-1} and a_i^{-1}, rather than the entire array. As f_i^{-1} and a_i^{-1} are both polynomials in $\mathcal{R}/q$, they require $\lceil \log_2 q \rceil$ bits per coefficient, as do the polynomials a_i. Thus, not having to store all a_i^{-1} and f_i^{-1} reduces the memory cost by almost two-thirds. The main memory footprint is now from the n polynomials a_i, which all have to be stored for the entire duration of the batch inversion.

6.7. Modular Reduction With and Without DSPs

In many parts during the Streamlined NTRU Prime KEM, we must compute the modular reduction of an integer. Examples are reducing polynomial coefficients to $\mathbb{Z}/3$, $\mathbb{Z}/q$, an

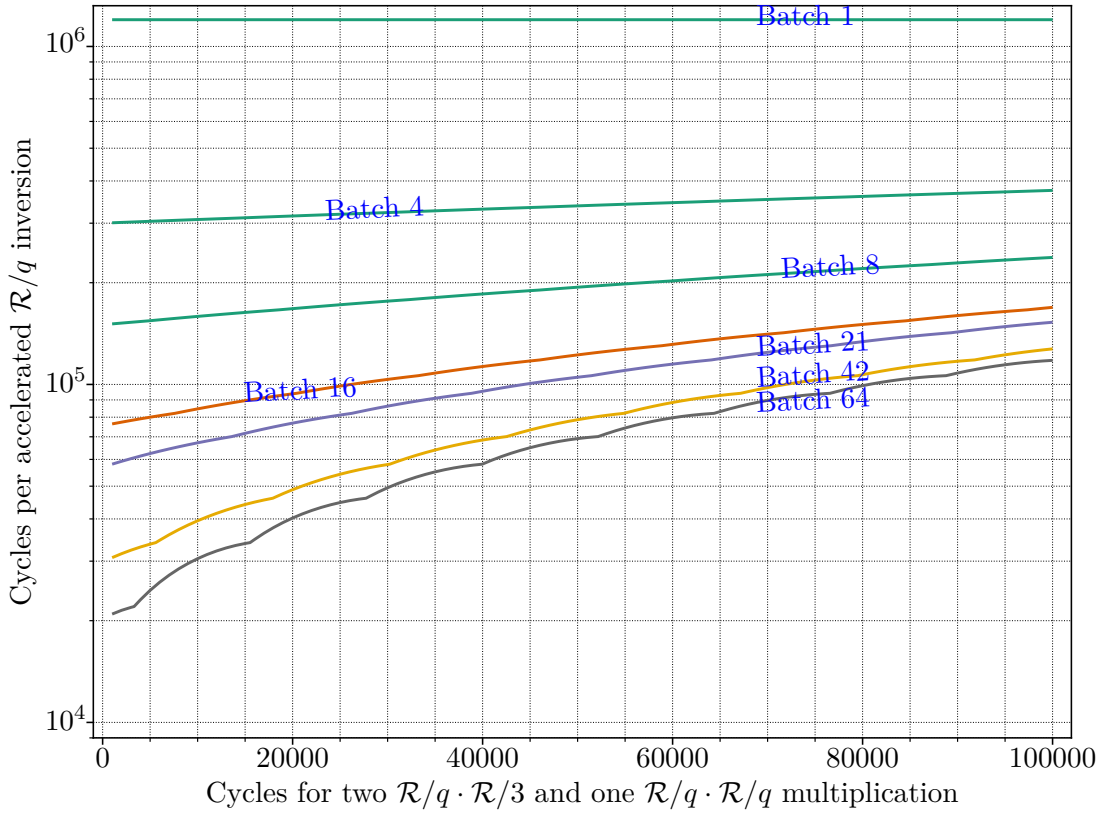


Figure 6.5.: Minimum batch size when comparing the cycle count for the three multiplications incurred per polynomial inversion when using Montgomery's trick, to simply accelerating the inversion itself [17]. This assumes a base $\mathcal{R}/q$ inversion speed of 1 200 000 cycles, which is roughly the number of cycles an $\mathcal{R}/q$ inversion takes with an unroll factor of 1 (i.e., no loop unrolling).

NTT-friendly prime, or during the decoding. A modulo 3 reduction is also needed for the rounding of ciphertexts during the encapsulation and re-encryption, and for address computation within the NTT module. Two methods are presented for such reductions: A DSP-based Barrett reduction, and a DSP free reduction based on LUT.

6.7.1. Barrett Reduction

Barrett reduction allows a modulo operation to be replaced with a multiplication with a precomputed constant and a bit shift [192]. This can be implemented efficiently in the DSP units of the FGPA. For the reduction to $\mathbb{Z}/3$, we slightly modify the constants from Barrett's original paper. The modified constants are needed to reduce values from a larger interval than normally allowed: Barrett reduction for a modulus q is only correct for the interval $[0, q^2]$. This obviously does not work if we wish to reduce values from $\mathbb{Z}/q$ to $\mathbb{Z}/3$ if $q > 9$. In addition, the modifications allow us to reuse the reduction to

```
import math
q=4591; q_half = math.floor(q/2); p=761
k=16; r=math.floor((2**k) / 3)
def reduce_and_round(input):
   rounded_output = 3*(((input+q_half) * r + 2**(k - 1)) >> k)-
   q_half
   mod_3_result = input - rounded_output
   return [rounded_output, mod_3_result]
```

Listing 6.2: A python version of our combined Barrett reduction to $\mathbb{Z}/3$ and rounding algorithm.

round coefficients to the nearest multiple of three. A Python version of our reduction algorithm with the constants is found in Listing 6.2. To verify the correctness of the modified constants, we mechanically verify in a simulator that all possible elements from $\mathbb{Z}/q$ are both rounded correctly and reduced to the correct values in $\mathbb{Z}/3$. This would mean that for elements outside of $\mathbb{Z}/q$, one would first have to reduce to $\mathbb{Z}/q$, and then again to $\mathbb{Z}/3$. However, this situation does not occur in Streamlined NTRU Prime.

Our Streamlined NTRU Prime implementation from [10] exclusively uses Barrett reduction for all modular reduction computations. Our implementation from [17] uses Barrett reduction in the decoder and the rounding module. The improved high-speed implementation only uses Barrett reduction in the decoder. For all other reductions, the DSP-free reduction is used, which is presented in the next section.

6.7.2. Reduction Without DSPs

The following work on modular reduction without DSP was primarily done by the coauthors of the Academia Sinica and the National Taiwanese University for our joint paper [17]. We extend the technique of fast modulo reduction in [104] (called *Shifting Reduction*) which does not use additional DSP slices. We apply this technique in the cases $q \in \{7681, 12289, 15361\}$. Moreover, in the case $q = 4591$, another reduction technique (called *Linear Reduction*) will be introduced. Linear reduction will also be used for $q = 4621$ and $q = 5167$. All modular reductions are fully pipelined and can process one new operand per clock cycle.

6.7.2.1. Signed Modular Reduction on $q = 12289$

We start with the modification of the unsigned reduction with $q = 12289$ as introduced in [104]. In the signed case, the reduction is slightly different: Suppose $-6144 \leq a \leq 6144$ and $-6144 \leq b \leq 6144$. We know that $z = ab$ is a 27 bit signed number (and not 28 bit as in the unsigned case) and

$$(5C00000)_{16} = -37748736 \leq z = ab \leq 37748736 = (2400000)_{16} \tag{6.5}$$

We have $q = 2^{14} - 2^{12} + 1$, so $2^{14} \equiv 2^{12} - 1 \pmod{q}$. The sign bit $z[26]$ contributes $-2^{26} \equiv 1365 = 2^{11} - 683 \pmod{q}$. With a similar derivation in [104], z can be re-

expressed as:

$$\begin{aligned}
z &= -2^{26}z[26] + 2^{14}z[25:14] + z[13:0] \\
&\equiv 1365z[26] + z[13:0] \\
&\quad + 2^{12}(z[25:24] + z[23:22] + z[21:20] + z[19:18] \\
&\qquad + z[17:16] + z[15:14]) \\
&\quad - (z[25:14] + z[25:16] + z[25:18] + z[25:20] \\
&\qquad + z[25:22] + z[25:24]) \\
&\equiv 2^{11}z[26] + z[11:0] \\
&\quad + 2^{12}(z[25:24] + z[23:22] + z[21:20] \\
&\qquad + z[19:18] + z[17:16] + z[15:14] + z[13:12]) \\
&\quad - (683z[26] + z[25:14] + z[25:16] \\
&\qquad + z[25:18] + z[25:20] + z[25:22] + z[25:24])
\end{aligned} \tag{6.6}$$

We separate the positive and negative terms and define:

$$\begin{aligned}
z_{pu} &\triangleq z[25:24] + z[23:22] + z[21:20] + z[19:18] \\
&\qquad + z[17:16] + z[15:14] + z[13:12] \\
z_{p^2u} &\triangleq z_{pu}[4] + z_{pu}[3:2] + z_{pu}[1:0] \\
z_{p^3u} &\triangleq z_{p^2u}[2] + z_{p^2u}[1:0] \\
z_p &\triangleq 2^{12}z_{pu} + 2^{11}z[26] + z[11:0] \\
z_n &\triangleq 683z[26] + z[25:14] + z[25:16] + z[25:18] \\
&\qquad + z[25:20] + z[25:22] + z[25:24]
\end{aligned} \tag{6.7}$$

Clearly $z_p - z_n \equiv z \pmod q$. z_{pu} is not greater than 21, z_{p^2u} is not greater than 6, and z_{p^3u} not greater than 3. z_p can be represented as

$$z_p \equiv 2^{12}z_{p^3u} + 2^{11}z[26] + Z[11:0] - (z_{p^2u}[2] + z_{pu}[4] + z_{pu}[4:2]) \triangleq z_p^* \pmod q \tag{6.8}$$

which is not greater than $12288 + 2048 + 4095 = 18431 < q + \frac{q-1}{2}$. We also have $z_n \leq 683 + (3 + 15 + 63 + 255 + 1023 + 4095) = 6074 < \frac{q-1}{2}$. Now $z_0 \triangleq z_p^* - z_n \equiv z \pmod q$ and is an integer in $[-6074, 18431]$. We need only to check if z_0 is greater than $\frac{q-1}{2} = 6144$, and perform a subtraction of q if this is the case. The equivalent logic circuit is given in Figure 6.6. The bold blocks and dataflows differ from that in [104] due to our use of signed reduction.

6.7.2.2. Signed Modular Reduction on $q = 7681$

Shifting Reduction can be easily applied in the case $q = 7681$ since q is of the form $q = 2^h - 2^l + 1$. Suppose $-3840 \leq a \leq 3840$ and $-3840 \leq b \leq 3840$. Now $z = ab$ is a 25 bit signed number and

$$(11\text{F}0000)_{16} = -14745600 \leq z = ab \leq 14745600 = (0\text{E}10000)_{16} \tag{6.9}$$

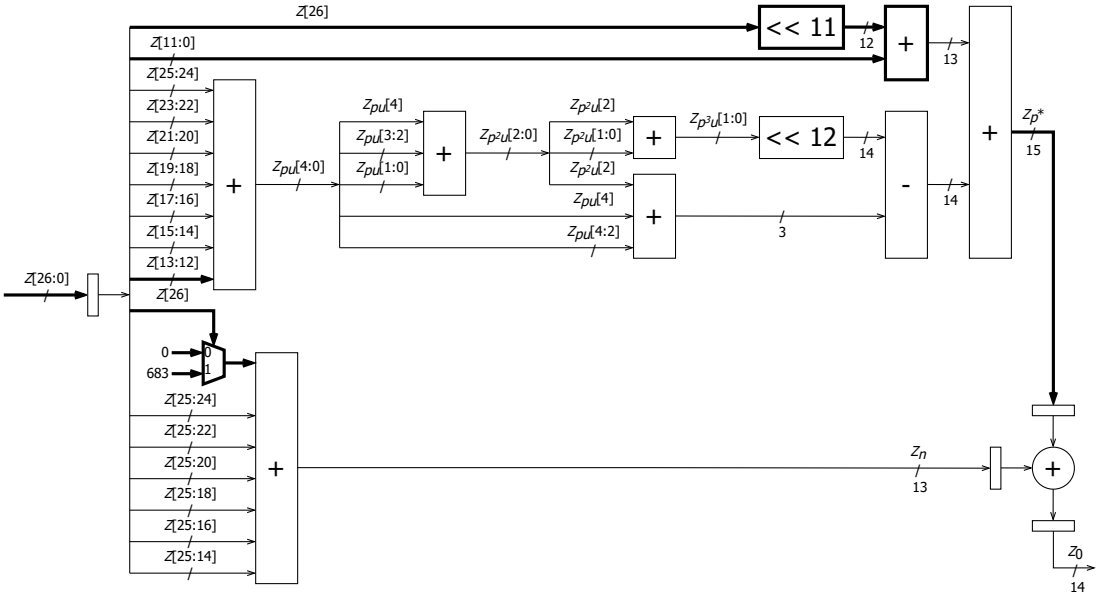


Figure 6.6.: Modified circuit for signed reduction modulo 12289 [17].

Since $q = 2^{13} - 2^9 + 1$, we have $2^{13} \equiv 2^9 - 1 \pmod{q}$. The sign bit $z[24]$ contributes $-2^{24} \equiv -1912 \pmod{q}$. Now z can be re-expressed as

$$\begin{aligned} z &= -2^{24} z[24] + 2^{13} z[23:13] + z[12:0] \\ &\equiv z[12:0] + 2^9 (z[23:21] + z[20:17] + z[16:13]) \\ &\quad - (1912 z[24] + z[23:21] + z[23:17] + z[23:13]) \\ &\equiv z[8:0] + 2^9 (z[23:21] + z[20:17] + z[16:13] + z[12:9]) \\ &\quad - (1912 z[24] + z[23:21] + z[23:17] + z[23:13]) \pmod{q} \end{aligned} \tag{6.10}$$

We define:

$$\begin{aligned} z_{pu} &\triangleq z[23:21] + z[20:17] + z[16:13] + z[12:9] \\ z_{p^2u} &\triangleq z_{pu}[5:4] + z_{pu}[3:0] \\ z_{p^3u} &\triangleq z_{p^2u}[4] + z_{p^2u}[3:0] \\ z_p &\triangleq z[8:0] + 2^9 z_{pu} \\ z_n &\triangleq 1912 z[24] + z[23:21] + z[23:17] + z[23:13] \end{aligned} \tag{6.11}$$

z_{pu} is a 6 bit unsigned integer, and if $z_{pu}[5:4] = 3$, then $z_{pu}[3:0] \leq 4$ and $z_{p^2u} \leq 7$. So $z_{p^2u} \leq 17$ and is a 5 bit integer and then $z_{p^3u} \leq 15$, and $z_{p^2u}[4] + z_{pu}[5:4] \leq 3$. Now

$$\begin{aligned} z_p &= z[8:0] + 2^9 z_{pu} \\ &\equiv z[8:0] + 2^9 z_{p^2u} - z_{pu}[5:4] \\ &\equiv z[8:0] + 2^9 z_{p^3u} - (z_{p^2u}[4] + z_{pu}[5:4]) \triangleq z_p^* \pmod{q} \end{aligned} \tag{6.12}$$

and is bounded by 8191. On the other hand, z_n is bounded by $1912 + 7 + 127 + 2047 = 4093$. Therefore, $z_0 \triangleq z_p^* - z_n \equiv z_p - z_n = z \pmod{q}$ and is an integer in $[-4093, 8191]$. We can tighten the possible values to $[-3581, 8191]$ because of following lemma:

Lemma 6.7.1. *$z_0 \geq -3581$, which is larger than $-(q-1)/2 = -3840$.*

Proof. We only need to consider the case where $3582 \leq z_n \leq 4093$, since for the case $z_n < 3582$ the inequality always holds. Using Equation 6.11 we can write:

$$3582 \leq 1912z[24] + z[23:21] + z[23:17] + z[23:13] \leq 4093 \tag{6.13}$$

We know that the bound of each term is 1912, 7, 127 and 2047 respectively. To ensure that z_n is not less than 3582, we need $z[24] = 1$. We can also place a bound on $z[23:13]$:

$$3582 - 1912 - 7 - 127 = 1539 = (603)_{16} \leq z[23:13] \leq 2047 = (7\text{FF})_{16} \tag{6.14}$$

Then because $z[23:21] \geq 6$, it follows that $z_{pu} \geq 6$, and further $z_{p^2u} \geq 1$ and $z_{p^3u} \geq 1$. Therefore $z_p^* \geq 512$, which in turn ensures

$$z_p^* - z_n \geq 512 - 4093 = -3581. \tag{6.15}$$

□

We now only need to determine if $z_0 > 3840$, in which case another signed subtraction by q is necessary to bound the result in $[-3840, 3840]$. The circuit for the signed reduction modulo 7681 is shown in Figure 6.7. We can see that the architecture is very similar to that for modulo 12289. The main difference is the dataflow of the sign bit.

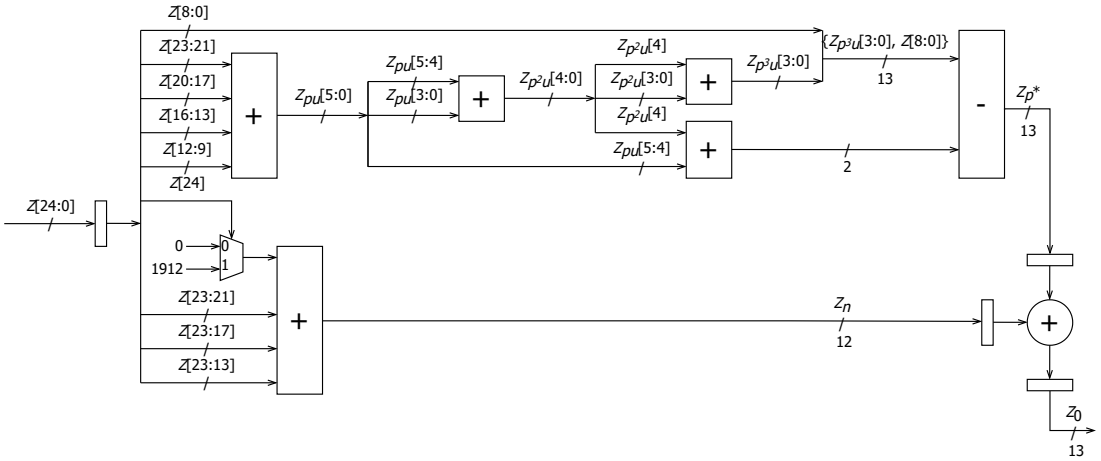


Figure 6.7.: Equivalent circuit for signed reduction modulo 7681 [17].

6.7.2.3. Signed Modular Reduction on $q = 15361$

We can also use Shifting Reduction in the case $q = 15361$. Suppose $-7680 \leq a \leq 7680$ and $-7680 \leq b \leq 7680$. Now $z = ab$ is a 27 bit signed number and

$$(47C0000)_{16} = -58982400 \leq z = ab \leq 58982400 \leq (3840000)_{16} \tag{6.16}$$

We have $q = 2^{14} - 2^{10} + 1$, and the sign bit $z[26]$ contributes $-2^{26} \equiv 3345 = 2^{12} - 751 \pmod{q}$. z can be re-expressed as

$$\begin{aligned} z &= -2^{26}z[26] + 2^{14}z[25:14] + z[13:0] \\ &\equiv 3345z[26] + z[13:0] + 2^{10}(z[25:22] + z[21:18] + z[17:14]) \\ &\quad - (z[25:14] + z[25:18] + z[25:22]) \\ &= 2^{12}z[26] + z[9:0] \\ &\quad + 2^{10}(z[25:22] + z[21:18] + z[17:14] + z[13:10]) \\ &\quad - (751z[26] + z[25:14] + z[25:18] + z[25:22]) \end{aligned} \tag{6.17}$$

We define:

$$\begin{aligned} z_{pu} &\triangleq z[13:10] + z[25:22] + z[21:18] + z[17:14] \\ z_{p^2u} &\triangleq z_{pu}[5:4] + z_{pu}[3:0] + 4z[26] \\ z_{p^3u} &\triangleq z_{p^2u}[4] + z_{p^2u}[3:0] \\ z_p &\triangleq 2^{10}z_{pu} + 2^{12}z[26] + z[9:0] \\ z_n &\triangleq 751z[26] + z[25:14] + z[25:18] + z[25:22] \end{aligned} \tag{6.18}$$

Note that the definition of z_{p^2u} is slightly different from the other cases. We can see that z_{pu} is a 6 bit unsigned integer. If $z_{pu}[5:4] = 3$, then $z_{pu}[3:0] \leq 12$ and $z_{p^2u} \leq 19$. This means that $z_{p^2u} \leq 21$ and is a 5 bit integer. Now

$$\begin{aligned} z_p &= 2^{10}z_{pu} + 2^{12}z[26] + z[9:0] \\ &= 2^{10}z_{pu}[3:0] + 2^{14}z_{pu}[5:4] + 2^{10} \cdot 4z[26] + z[9:0] \\ &\equiv z[9:0] + 2^{10}z_{p^2u} - z_{pu}[5:4] \\ &\equiv z[9:0] + 2^{10}z_{p^3u} - (z_{p^2u}[4] + z_{pu}[5:4]) \triangleq z_p^* \pmod{q} \end{aligned} \tag{6.19}$$

and is bounded by 16383. z_n is bounded by $751 + 4095 + 255 + 15 = 5116$. Therefore, $z_0 = z_p^* - z_n \equiv z_p - z_n = z \pmod{q}$ and is an integer in $[-5116, 16383]$. We need only to check if the value of z_0 is greater than 7680 and perform a subtraction of q if this is the case. The circuit diagram for signed reduction modulo 15361 is omitted as it is similar to those for modulo 7681 and 12289.

6.7.2.4. Signed Modular Reduction on $q = 4591$

The reduction of integers modulo $q = 4591$ (or other q's of the parameter sets of Streamlined NTRU Prime) using Shifting Reduction is not easily obtained since all the primes

are not of the form $q = 2^h - 2^l + 1$. For example, $q = 4591 = 2^{12} + 2^9 - 2^4 - 1$ is of effective Hamming weight 4. Shifting Reduction will make the bits spread into the lower bits, making the positive and the negative parts of the partial results (as z_p and z_n defined in the case $q \in \{7681, 12289, 15361\}$) hard to be analyze.

In the signed version modification doing modulo 12289, we separate the sign bit from other bits and consider it independently. However, we can actually consider each individual bit independently. This allows us to transform the reduction problem into several signed additions. We will call this technique *Linear Reduction*. This technique was first presented in [17], for $q = 4591$ for the `sntrup761` parameter set.

In our implementation, the integer z that will be reduced is a 33 bit signed integer bounded by

$$\begin{aligned}(11117\text{A}137)_{16} \leq & z \leq (0\text{EEE85EC9})_{16} \\ -4008206025 \leq & z \leq 4008206025 = (2295)^2 \cdot 761\end{aligned} \tag{6.20}$$

In this case z can be represented as

$$\begin{aligned}z &= -2^{32} z[32] + \sum_{i=0}^{31} 2^i z[i] \\ &\equiv 433 z[32] \\ &+ 2079 z[31] + 3335 z[30] + 3963 z[29] + 4277 z[28] + 4434 z[27] \\ &+ 2217 z[26] + 3404 z[25] + 1702 z[24] + 851 z[23] + 2721 z[22] \\ &+ 3656 z[21] + 1828 z[20] + 914 z[19] + 457 z[18] + 2524 z[17] \\ &+ 1262 z[16] + 631 z[15] + 2611 z[14] + 3601 z[13] + 4096 z[12] \\ &+ z[11:0] \pmod q\end{aligned} \tag{6.21}$$

We could do 22 signed modular additions, but this approach would make the critical path be 5 signed modular additions, assuming we use a tree-adder like structure. With an implementation in hardware, we can actually pre-combine some of the additions.

The basic idea is to utilize the power of Look-Up Tables. Xilinx FPGAs provide LUT units supporting the functions of both $\text{LUT}_{5,2}$ and $\text{LUT}_{6,1}$. We can divide the most significant 21 bits into 5 groups, each containing 3 to 5 specified bits, and collect $z[11:0]$ as one group. Specifically, we define

$$\begin{aligned}p_0 &\triangleq z[11:0] \\ p_1 &\triangleq (3335 z[30] + 2721 z[22] + 2524 z[17] + 2611 z[14]) \bmod q \\ p_2 &\triangleq (433 z[32] + 851 z[23] + 914 z[19] + 457 z[18] + 631 z[15]) \bmod q \\ n_0 &\triangleq -(3963 z[29] + 4277 z[28] + 3404 z[25] + 3656 z[21] + 3601 z[13]) \bmod q \\ n_1 &\triangleq -(4434 z[27] + 1262 z[16] + 4096 z[12]) \bmod q \\ n_2 &\triangleq -(2079 z[31] + 2217 z[26] + 1702 z[24] + 1828 z[20]) \bmod q\end{aligned} \tag{6.22}$$

We could apply any other partition, however the partition we decide to use is beneficial for the later calculations because we can bound each group:

$$\begin{aligned} p_0 &\leq 4095 \\ p_1 &\leq 4076 < 4095 \\ p_2 &\leq 3286, p_2 + 4591 \leq 7877 < 8191 \\ n_0 &\leq 4054 < 4095 \\ n_1 &\leq 3981 < 4095 \\ n_2 &\leq 3573, n_2 + 4591 \leq 8164 < 8191 \end{aligned} \tag{6.23}$$

All possible values of p_1, p_2, n_0, n_1, and n_2 are pre-calculated and stored in the distributed memory constructed by $\text{LUT}_{5,2}$ units. The values of p_1, p_2, n_0, n_1, and n_2 are then determined at the outputs of the LUTs, according to the inputs $z[33:12]$.

Now we can easily implement the reduction with the modular additions:

$$\begin{aligned} p_{01} &\triangleq p_0 + p_1 \\ n_{01} &\triangleq n_0 + n_1 \\ z_p &\triangleq (p_{01} \bmod q) + p_2 \\ z_n &\triangleq (n_{01} \bmod q) + n_2 \\ z \equiv z^* &\triangleq z_p - z_n \end{aligned} \tag{6.24}$$

We can see that p_{01}, n_{01}, z_p, z_n are all 13 bit unsigned integers. and z^* is bounded by $[-8191, 8191]$, which is a 14 bit signed integer. $z \bmod^{\pm} q$ can be found by

$$z \bmod^{\pm} q = z^* + kq, k \in \{-2, -1, 0, 1, 2\} \tag{6.25}$$

6.7.2.5. Signed Modular Reduction on $q = 4621$ and $q = 5167$

We can apply linear reduction in the same way as in Section 6.7.2.4 to the other moduli of the other Streamlined NTRU Prime parameter sets. Concretely, we also apply the technique to $q = 4621$ and $q = 5167$, which are needed for the sntrup653 and sntrup857 parameter sets respectively. The reduction module for $q = 4621$ was implemented by Bo-Yuan Peng of the Academia Sinica, while I implemented the reduction module for $q = 5167$. Because the linear reduction is applied in an identical way, we leave out the details of the partitioning.

6.7.2.6. Signed Modular Reduction on $q = 3$

For the modular 3 reduction, we can use a related technique as above to avoid needing a DSP. This technique was first presented in [20]. During the reduction, we have an

input from $\mathbb{Z}/q$ and want to reduce it to $\mathbb{Z}/3$, specifically $\{-1, 0, 1\}$. We start with an unsigned 13 bit number $z[12:0]$ and repeatedly exploit the relation $2 \equiv -1 \bmod 3$.

$$\begin{aligned} z[12:0] &\equiv 2z[12:1] + z[0] \equiv -z[12:1] + z[0] \bmod 3 \\ &\equiv -2z[12:2] - z[1] + z[0] \equiv z[12:2] - z[1] + z[0] \equiv \ldots \bmod 3 \\ &\equiv \sum_{i=0}^{6} z[2i] - \sum_{i=0}^{5} z[2i+1] \bmod 3 \end{aligned} \tag{6.26}$$

The result of this computation ranges from -6 to 7 and is represented by a signed 4 bit integer $y[3:0] = -2^3 y[3] + y[2:0]$. We again exploit the above relation:

$$\begin{aligned} -2^3 y[3] + y[2:0] &\equiv y[3] + y[2:0] \equiv y[3] + 2y[2:1] + y[0] \bmod 3 \\ &\equiv y[3] - y[2:1] + y[0] \equiv y[3] - 2y[2] - y[1] + y[0] \bmod 3 \\ &\equiv y[3] + y[2] - y[1] + y[0] \bmod 3 \end{aligned} \tag{6.27}$$

This results in a value ranging from -1 to 3, represented by a signed 3 bit integer $x[2:0] = -4x[2] + x[1:0] \equiv x[1:0] - x[2] \bmod 3$. This value can already be mapped to a value $w[1:0] \in \{-1, 0, 1\}$ efficiently:

$$w[0] = x[0] \oplus x[1] \oplus x[2] \tag{6.28}$$

$$w[1] = (x[0] \wedge x[1]) \oplus x[1] \oplus x[2] \tag{6.29}$$

One additional point to consider is that this modulo 3 calculation assumes an unsigned 13 bit number. However, in the Streamlined NTRU Prime specification, the modulo 3 operation is used on signed 13 bit numbers, in the interval $[-(q-1)/2, (q-1)/2]$ [9]. This means that numbers where $z[12] = 1$ (i.e., in the interval $[-(q-1)/2, -1]$) must be treated slightly differently, as these are negative. However, the solution is simple: since $q = 4591$, and $4591 = 1 \bmod 3$, we simply have to add 1 to the final result if $z[12] = 1$. This relation is true for all q in all Streamlined NTRU Prime parameter sets. When we want to perform the rounding, we first compute the input modulo 3, and then subtract that result from the input to receive the rounded input. All of the above we implement in only a small amount on LUT, together with some FF for pipelining.

6.8. Implementation of the Codec for Polynomials in $\mathcal{R}/3$ and $\mathcal{R}/q$

In order to save bytes during transmission and key storage, as well as to be able to hash polynomials, Streamlined NTRU Prime has specified an encoding for all the various polynomials. This encoding transforms the polynomials to and from byte strings. The polynomials of the private key f, g^{-1} and the secret r are all in $\mathcal{R}/3$. For these elements, each polynomial coefficient is in $\mathbb{Z}/3$ and can be represented with 2 bits. The encoding is thus trivial: Four coefficients are packed in a shift register into a single byte for the encoding (and vice-versa for the decoding). Because p is not a multiple of four in any of the parameter sets, the final byte is zero padded.

The codec for polynomials in $\mathcal{R}/q$ is more complicated, and uses the algorithms described in Section 4.5. The first hardware implementation to use these codecs is [10], as part of this thesis. The encoder can be implemented in a relatively straightforward way according to Listing 4.1. However, the decoder is significantly more complex. A reason for this is that the decoder requires a 32 bit by 16 bit division (Line 19 in Listing 4.2). In order to avoid the need to implement a full division circuit, we precalculate all divisors, which are not dependent on any secrets, and store these in a table. For $p = 761$ and $q = 4591$, there are in total 42 different divisors, each fitting in 16 bit. Half of these are for the decoding the rounded coefficients of the ciphertext, the other half are for the public key. Due to the precalculation, we can then use integer division by a constant, allowing us to replace the division with a multiplication and a bit shift [193].

However, our encoder and decoder from [10] are comparatively expensive and inefficient. An improved codec is presented in [17], also as a part of this thesis, though the implementation was primarily done by my coauthors from the Academia Sinica. For this implementation, we inspect inductively the details of the process of the encoder and decoder, especially how R_2 and M_2 (denoted as R2 and M2 in Listing 4.2) change with respect to R and M, how many output bytes there are in each round, and what exactly M_2 is during each recursive call. We will use $\langle\cdot\rangle$ to indicate the contents of the lists.

Case 1: When $\text{len}(M) = 1$, that is, $R = \langle r_0\rangle$ and $M = \langle m_0\rangle$, there is no recursive call. We know that $r_0 < 16384$, so all the bytes of r_0 are dumped as output bytes to the encoded sequence. If $m_0 > 255$, we output 2 bytes and otherwise output 1 byte.

Case 2: When $\text{len}(M) = 2$, that is, $R = \langle r_0, r_1\rangle$ and $M = \langle m_0, m_1\rangle$, we compute $r'_0 = r_0 + m_0 r_1$. The upper bound for r'_0 and the new m'_0 can actually be pre-determined just from m_0 and m_1. Whether 0, 1, or 2 bytes are sent as output can also pre-determined from m_0 and m_1.

Case 3: When $R = \langle r_0, ...r_{2n-1}, r_{2n}\rangle$, $M = \langle m_0, ..., m_0, m_1\rangle$ and $\text{len}(M) = 2n + 1$ where n is a positive integer, we can compute $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \le i \le n-1$. The upper bound of each r'_i and the new m'_0 can be pre-determined just from m_0. Whether 0, 1, or 2 bytes are sent as output can also pre-determined by m_0. We denote the "replaced" r'_i appended into R_2 as $r_i'^{(\text{replaced})}$, which satisfies

$$r_i'^{(\text{replaced})} \in \{r'_i, \lfloor\frac{r'_i}{256}\rfloor, \lfloor\frac{r'_i}{65536}\rfloor\} \tag{6.30}$$

and then we have $R_2 = \langle r_0'^{(\text{replaced})}, ..., r_{n-1}'^{(\text{replaced})}, r_{2n}\}$ and $M_2 = \langle m'_0, ...m'_0, m_1\rangle$, with $\text{len}(M_2) = n+1$. Note that the structure of M' and M are similar: a sequence of specified integers m_0 or m'_0 in $[1, 16383]$ followed by an integer m_1, which is either distinct from or the same as m_0 or m'_0.

Case 4: When $R = \langle r_0, ...r_{2n}, r_{2n+1}\rangle$, $M = \langle m_0, ..., m_0, m_1\rangle$ and $\text{len}(M) = 2n + 2$ where n is a positive integer, we can compute $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \leq i \leq n$. If $0 \leq i \leq n-1$, the upper bound of r'_i and the new m'_0 can be pre-determined only by m_0. The upper bound of r'_n, which is the last element in R_2, and the new m'_1, which is the last element in M_2, is pre-determined by both m_0 and m_1. For $0 \leq i \leq n-1$, whether 0, 1, or 2 bytes are sent as output when computing r'_i is also pre-determined by m_0. Whether 0, 1, 2 bytes are sent as output when computing r'_n is pre-determined by m_0 and m_1. We can also compute the resulting $R_2 = \langle r_0'^{(\text{replaced})}, ..., r_{n-1}'^{(\text{replaced})}, r_n'^{(\text{replaced})}\}$, $M_2 = \langle m'_0, ..., m'_0, m'_1\}$, and $\text{len}(M) = n + 1$. The structure of M_2 and M are still similar: a sequence of specified integers m_0 or m'_0 followed by an integer m'_1, which is either distinct from or the same as m_0 or m'_0.

We know that when the encode starts, $M = \langle q, ..., q\rangle$ and $\text{len}(M)$ is odd. This implies that we only need to track m_0, m_1 and the output bytes for each regular pair of r's and for the last r. Table 6.1 shows the values of m_0, m_1, and the number of output bytes. We can see that the total encoded bytes are of length 1158. For the rounded encode, we replace the q in M with $q' = 1531 = \lceil q/3 \rceil$. We can then also precompute similar tracking info, shown in Table 6.2. The total encoded bytes there are of length 1007.

Table 6.1.: Round information doing $\mathcal{R}/q$-encode for the parameter set sntrup761 [17].

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	761	4591	2	760	4591	N/A
2	381	322	1	190	4591	N/A
3	191	406	1	95	4591	N/A
4	96	644	1	47	4591	1
5	48	1621	1	23	11550	2
6	24	10265	2	22	286	1
7	12	1608	1	5	11468	2
8	6	10101	2	4	282	1
9	3	1557	1	1	11127	N/A
10	2	9740	N/A	N/A	11127	2
11	1	N/A	N/A	N/A	1608	2

The block diagrams of the encoder and decoder are shown in Figure 6.8a and 6.8b, where the dashed blocks are external modules. The parameter module with the round information is a look-up table of either Table 6.1 or Table 6.2, allowing the encoder and decoder to flexibly switch between either $\mathcal{R}/q$-encode or rounded-encode (respectively decode). In addition, this also allows the circuits to be able to perform the encode/decode for any parameter set, as one only has to replace the parameter module. The encoder

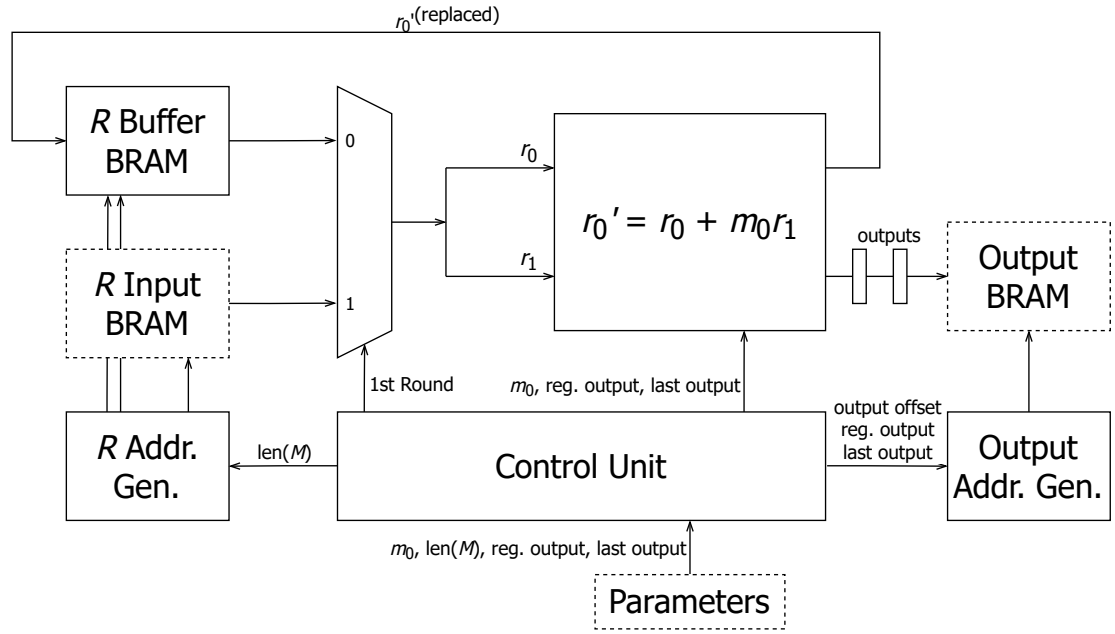
Table 6.2.: Round information doing rounded encode for the parameter set sntrup761 [17].

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	761	1531	1	380	1531	N/A
2	381	9157	2	380	1531	N/A
3	191	1280	1	95	1531	N/A
4	96	6400	2	94	1531	2
5	48	625	1	23	150	1
6	24	1526	1	11	367	1
7	12	9097	2	10	2188	2
8	6	1263	1	2	304	1
9	3	6232	2	2	1500	N/A
10	2	593	N/A	N/A	1500	1
11	1	N/A	N/A	N/A	3475	2

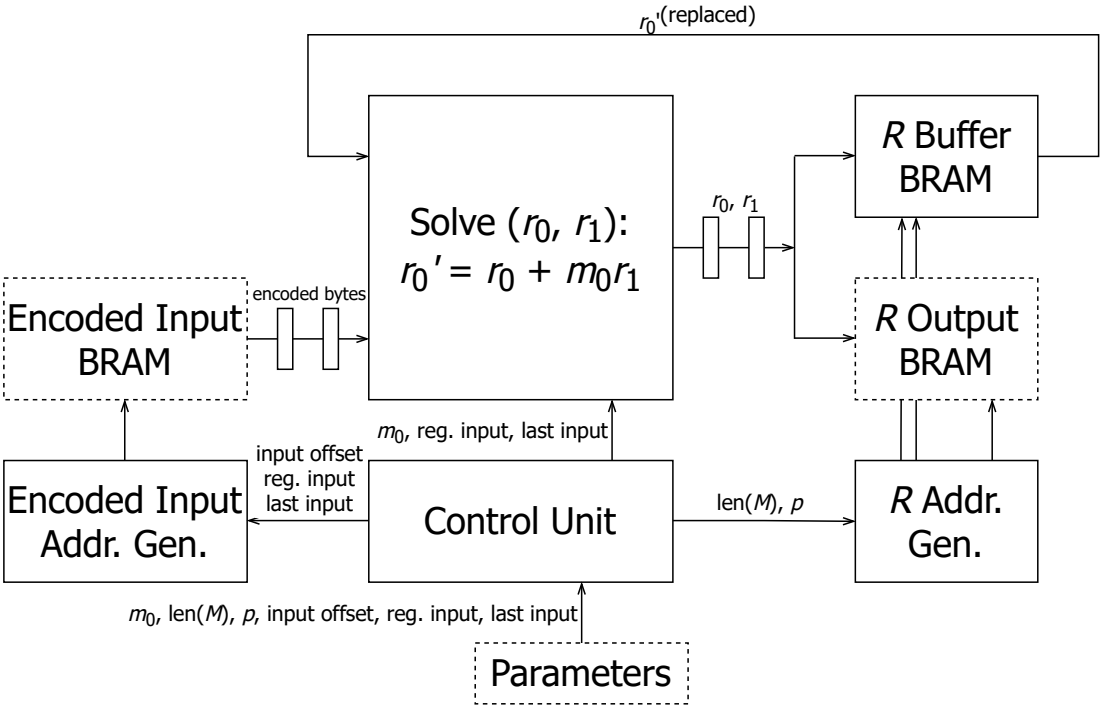
needs a DSP slice to evaluate $r'_0 = r_0 + m_0 r_1$. The decoder needs 4 DSP slices to apply Barrett's reduction to evaluate $r_0 = r'_0 \mod m_0$. Both of the encoder and the decoder also need an internal memory buffer to save the intermediate R values.

To improve our work from [17], we carefully analyze and optimize the register pipelining, in particular of the Barrett reduction in the decoder. This allows us to reduce the critical path and in turn increase the maximum clock frequency. A further improvement of the encoder lies in the retrieval of the r_0 and r_1 values. In [17], one word of the R input BRAM (or R buffer BRAM) is fetched per clock cycle. Hence, it takes two clock cycles before $r'_0 = r_0 + m_0 r_1$ can be evaluated. A straightforward optimization is to read two words per clock cycle from the BRAMs per clock cycle, allowing $r'_0 = r_0 + m_0 r_1$ to be evaluated every clock cycle. This requires both the external R input BRAM and the buffer BRAM to be able to deliver two polynomial coefficients per clock cycle. It does not require any changes to the round information in the parameter modules. All of these improvements were done by me.

In [17], we presented the parameter modules for all en- and decoding for the Streamlined NTRU Prime parameter set sntrup761. We extend this in this thesis with the support of all en- and decoding for both the sntrup653 and sntrup857 parameter set. The parameter module for sntrup653 was created by Bo-Yuan Peng from the Academia Sinica, while I created the parameter module for sntrup857. The tables with the corresponding precomputed round information are given in Appendix A.1.



(a) The encoder.



(b) The decoder.

Figure 6.8.: The block diagram of the encoder and decoder [17].

6.9. Weight Check

During the decapsulation, after the core decryption, the weight of the secret polynomial r' is checked (Line 6 in Algorithm 3). It is supposed to have a weight of w. If r' does not have a weight of w, it is replaced by a polynomial where the first w coefficients are 1, and the rest 0. The simplest way of implementing this weight check is by computing the sum of the absolute values of all coefficients. This sum should be exactly w.

Both of our designs in [10] and [17] implement the weight check in this way, with a serial adder: One coefficient is added per clock cycle to a counter, and afterwards the counter value is checked whether it is equal to w. The output of this check is then used to control a multiplexer to decide between the original r' and the replacement polynomial. However, the use of a dedicated $\mathcal{R}/3$ multiplier allows another approach: After the computation of the $\mathcal{R}/3$ multiplier is complete, the r' polynomial lies in the accumulator array. This means we can access the entirety of r' at the same time. Instead of adding up the weight of r' one coefficient per clock cycle, we use a parallel tree adder to sum up all coefficients in just 2 clock cycles. The sum is then compared to w and used as the control signal for the multiplexer.

6.10. SHA-512 Hash Function

Streamlined NTRU Prime uses SHA-512 internally as a hash function (see Section 4.6). It is used on the one hand to generate the shared secret after encapsulation and decapsulation, but also to create the confirmation hash. The confirmation hash is a hash of the public key and the short polynomial r and is appended to the ciphertext.

The SHA-512 implementation in our design from [10] is based on the open-source SHA-512 implementation from [194], with some modifications to improve performance and reduce resource consumption. Notably, rather than to precompute the entire message scheduled array, it is computed on the fly. The hashing of a 1 024 bit block takes 325 cycles. At the same time, the SHA-512 module is still relatively expensive, especially when compared to the small total time spent on hashing. This area cost can be reduced, in particular by storing the internal state variables in LUT RAM rather than FF. Our SHA-512 implementation from [17] is in turn also based on the implementation from [10]. There are some additional performance improvements, in the form that the majority of the round function updates now only takes a single clock cycle. As a result, the hashing of a 1 024 bit block takes 117 cycles.

We improve the SHA-512 module further, as especially for high-speed designs the SHA-512 hashing is becoming a bottleneck (see also Figure 7.7 and Section 7.5 in general). Apart from further improving the speed for a single 1 024 bit block (it now takes only 90 cycles), a significant improvement is the scheduling of the hash operations and the data input management. Multiple First-In-First-Out (FIFO) memories are used to buffer the

incoming to-be-hashed data, together with a command buffer that stores which FIFO should be hashed. This on the one hand prevents other modules in the design from stalling, and on the other hand allows the SHA-512 module to be active nearly the entire time, with little idling.

A further improvement is a higher maximum clock frequency. The critical path of SHA-512 in [17] are the ten 64 bit additions, of which seven are of the round function and three are from the message schedule array. These are computed in a single clock cycle. The critical path can be reduced to six additions, as the additions in the message schedule array, w_i and K_i can be computed independently of the 512 bit state. This allows us to compute $w_i + K_i$ in round $i - 1$, and then insert a pipeline stage to reduce the critical path. Finally, the additions were rewritten to make better use of the fast carry logic present in Xilinx FPGAs, again increasing the maximum clock frequency [79, 80].

6.11. Pre-Hashing of the Shared Secret and Rejection

A further significant improvement for the decapsulation is the *pre-hashing* of both the shared secret ss and the rejection ss'. During decapsulation, as part of the FO-transform, the ciphertext is recomputed and compared with the received ciphertext (see Line 9 to 14 in Algorithm 3). Depending on this check, either the actual shared secret ss is computed, or alternatively a pseudo-random value ss' is computed. However, neither ss nor ss' are dependent on the re-encryption step, or the output of the FO-transform. As a result, the SHA-512 module can pre-compute both values and store them. After the ciphertext comparison is complete, the SHA-512 module can output the correct hash value. Because ss' depends only on the received ciphertext and ρ from the private key, it can be computed immediately after the loading of the ciphertext and private key. To compute ss, the secret r' is required, so it can be computed after the core decryption is complete, in parallel to the re-encryption. Note that this type of pre-hashing is only possible as during Streamlined NTRU Prime decapsulation, the SHA-512 module is not used except to recompute the confirmation hash during the re-encryption and compute the shared secret. This leads to the SHA-512 module being idle for most of the decapsulation, allowing it to perform the pre-hashing.

Because the pre-hashing is still in constant time, it does not expose a side-channel vulnerability against simple SCA. However, in cases where we can expect more advanced attacks such as fault attacks or differential power analysis, pre-hashing should not be used.

6.12. Architecture

In the previous sections, various different individual hardware modules are described. These modules implement the core functionality of Streamlined NTRU Prime. We then assemble individual modules together to implement the full KEM. There, a number of

Finite State Machines (FSMs) are included to schedule the individual operations in the correct order, together with BRAM and FIFOs to store polynomials.

As a whole, the architectures of our designs in [10] and [17] are highly similar and have also remained the same for the improvements introduced in this thesis. First, modules which are needed across encapsulation, decapsulation and key generation are instantiated only once, and shared across operations. A central FSM controls which operation is currently being performed and routes the data accordingly. An exception to the sharing is the $\mathcal{R}/3$ encoder, which due to its small size is not shared. Also special is the FSM and memory of the core encryption. Since it is needed both during encapsulation, and the re-encryption during decapsulation, it is also shared. Modules that are unique to an operation are within the top-level module for the respective operation. Some modules are optional: For example, the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier is only included when batch inversion is used. The full architecture is displayed in Figure 6.9.

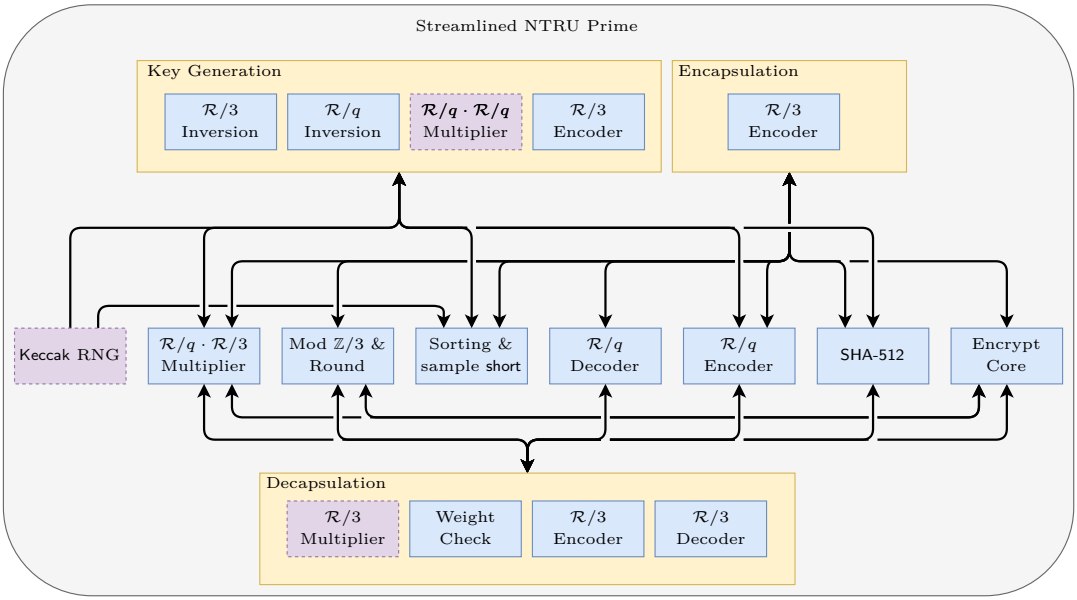


Figure 6.9.: Architecture of the Streamlined NTRU Prime KEM. This architecture is also used, with some minor differences, in [10] and [17]. Modules with dotted lines are optional.

An improvement in the architecture of the high-speed design in this work over our design in [17] is the better instruction scheduling and operation overlapping: In [17], the FSMs wait until the individual operations such as decoding, multiplication etc. are complete before starting the next operation. In some cases this is strictly necessary: For example, the weight check must be complete (and overwrite r' if needed) before the re-encryption step can proceed. However, in many other cases, the individual sub-operations can overlap, so that the output of one operation overlaps with the input of

another. We carefully analyze the instruction scheduling of the high-speed design for encapsulation and decapsulation in order to identify opportunities of such overlapping and use FIFOs memories when necessary to buffer input and output data. Furthermore, while our design in [17] uses two separate decoders for the public key and ciphertext during decapsulation, a careful scheduling allows us to make do with only a single decoder. The decoding of the ciphertext and public now occurs sequentially. Diagrams showcasing the operation scheduling for all operations can be found in Section 7.5. An additional improvement over [17] is our support for more parameter sets. In addition to sntrup761, we also support sntrup653 and sntrup857, though with one caveat: We do not support batch inversion for sntrup857, as the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier does not support the multiplication of polynomials of degree 857.

One part that is explicitly excluded in [10] and [17] is any sort of Random Number Generator (RNG). Instead, it is assumed that there is an external RNG that provides 32 bits of randomness per clock cycle. However, this somewhat distorts the area usage because generating very large amounts of cryptographically secure randomness can in of itself be expensive. As a result, we optionally add an internal RNG, based on the Keccak sponge construction [195]. Keccak has excellent performance when implemented in hardware, allowing us to generate large amounts of secure randomness based on a short seed. We use the Keccak implementation from [189], which is available on Github [196].

Table 6.3.: A summary of all of our high-speed and low-area implementations. A batch size of 1 indicates that no batch inversion is used.

Design	First published in	Supported parameters	Internal RNG	Batch Size	Loop Unroll Factor	
					$\mathcal{R}/3$ Inv.	$\mathcal{R}/q$ Inv.
Improved high-speed	This work	sntrup653	Yes	25	4	32
		sntrup761		21	4	32
		sntrup857		1	4	4
High-speed	[17]	sntrup761	No	21	4	32
Low-area	[17]	sntrup761	No	1	2	2
Low-area	[10]	sntrup761	No	1	1	1

In addition to a high-speed design, we also present a low-area implementation in [17]. The difference between the high-speed and the low area design lie in a number of different sub modules, though both use the same architecture described in this section. For one, the low-area version does not use batch inversion for key generation, and only uses an unroll factor of 2 instead of 4 during $\mathcal{R}/q$ inversion, and an unroll factor of 2 instead of 32 for the $\mathcal{R}/3$ inversion. The low-area implementation also uses the compact version of the parallel schoolbook multiplier, with only 24 MAC units rather than 761. Finally, the high-speed implementation in [17] uses two separate decoders, one for public keys,

and one for ciphertexts. This allows the private key (which also contains the public key) and the ciphertext to be decoded in parallel during decapsulation. In the low-area implementation, only one decoder is present, and the decoding occurs sequentially. Note that, due to an improvement in the decoder scheduling, we again need only a single decoder in our improved high-speed design. A summary of all of our implementations can be found in Table 6.3.

7. Evaluation & Discussion

In this chapter, we first compare and evaluate individual sub-modules in Section 7.1. We then compare and evaluate the full designs, for both high speed and low area, in Section 7.2. For both the sub-modules and the full designs we provide area utilization and performance results. Afterwards, we compare our designs with those of the literature in Section 7.3. As part of the comparison with the literature, we compare Core-SVP, latency and speed-area products in Section 7.4. Finally in Section 7.5, we provide and evaluate scheduling diagrams for the internal operations of our high-speed design and use these to highlight bottlenecks and sections for future improvements.

7.1. Evaluation of Sub-Modules

In this section, we will compare and evaluate the individual sub-modules that we developed as part of this thesis. Unfortunately, many works in the literature do not report the resource cost and benchmark numbers of their sub-modules. When these are available, we will compare our sub-modules with sub-modules from other works.

7.1.1. Sorting and Fixed-Weight Sampling

Table 7.1 shows a comparison of different sorting algorithm for arrays of size 761. Our radix sort from [17] is significantly faster than our sorting network from [10]. While not quite as fast as the FIFO merge sort from [88], the radix sort does use less LUT, FF and BRAM, and runs at a higher frequency. Thanks to improvements in the critical path in other modules, the radix sorting module can also run at a much higher frequency in this work. While this does lead to a slight increase in LUT and FF usage compared to [17], the latency is significantly improved, almost reaching the speed of the sorting module from [88], while still using less area.

Table 7.1.: A comparison of different sorting algorithms for generating fixed-weight short polynomials of degree 761 for NTRU and Streamlined NTRU Prime. The target FPGA is a Xilinx Zynq Ultrascale+.

Design	LUT	FF	BRAM	DSP	Freq. [MHz]	Cycles	Time [μs]
Radix sort, TW	844	285	2	0	400	4 842	12.11
Radix sort[1]	823	220	2	0	285	4 837	16.97
Sorting network[2]	231	87	1.0	0	279	49 400	177.1
FIFO merge sort [88]	1 441	940	3.5	0	250	2 762	11.05

[1] We first presented this implementation in [17]. [2] We first presented this implementation in [10].

Due to the pregeneration of short polynomials, the cycle count of the sorting does not factor into the cycle count of the encapsulation, as long as the encapsulation operation takes longer than the sorting. This was the case in [17]. However, to due further improvements in encapsulation (see Table 7.11), a single radix sorting module is no longer fast enough. This is solved by instantiating two radix sorting modules, which independently generate short polynomials. This also leads to a small speed-up in batch key generation, as it requires a number of short polynomials equal to the batch size.

7.1.2. Polynomial Multiplication

Table 7.2 show a comparison of different multiplication algorithms for Streamlined NTRU Prime. This includes our Karatsuba multiplier from [10], our high-speed and low-area schoolbook multiplier from [17], our NTT & CRT multiplier from [17], as well as the improved versions presented in this thesis. From the literature, we include the multipliers from [197] and [186].

Looking at our multipliers first, we can see that our two high-speed schoolbook multipliers from this work and [17] are by far the fastest, by over an order of magnitude. At the same, they are also by far the most resource intensive. The Karatsuba-based multiplier from [10] is the most compact with regards to LUT, but it also is the slowest, and has a comparatively high BRAM usage. The low-area schoolbook multiplier uses no BRAM, and only slightly more LUT, but is more than three time faster with regards to cycle count than the Karatsuba-based multiplier with a single recursion layer. The Karatsuba multiplier with two recursion layers (labeled "2x Karatsuba" in Table 7.2) has a similar cycle count as the low-area schoolbook multiplier, but also uses more than twice the LUT and FF, as well as a significant amount of BRAM. We conclude that the improved asymptotic complexity of the Karatsuba multiplier does not lead to a more efficient design, as the Karatsuba multiplier cannot exploit the small coefficients to the same degree as the pure schoolbook multipliers.

We see a similar situation with the NTT & CRT multiplier: The low-area schoolbook multiplier is better both in terms of resources consumed and cycle count and requires no BRAM or DSP. However, the NTT multiplier has the benefit of being extendable to perform $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication, with no increase in cycle count and only a moderate increase in resource consumption. As a result, we only use the NTT multiplier for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication during batch inversion. We use the same NTT multiplier for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication for both the sntrup653 and sntrup761 parameter set. Thus, the area consumption and latency are also identical for both parameter sets. This is different for the schoolbook multipliers, where LUT, FF and cycle count increase nearly proportionally to the polynomial degree of the larger parameter sets.

The high-speed schoolbook multiplier from this work has slightly lower LUT and FF usage than our previously published high-speed schoolbook multiplier in [17]. This is due to the fact the accumulator array in [17] was actually one bit wider than required.

Table 7.2.: A comparison of different multiplication algorithms for Streamlined NTRU Prime. The target FPGA is a Xilinx Zynq Ultrascale+. Entries labeled "TW" are new results from this thesis.

Design	Op	Param.	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
NTT & CRT, TW	$\mathcal{R}/q \cdot \mathcal{R}/q$	sntrup653	2 044	1 681	7.5	3	400	35 465	88.66
NTT & CRT, TW		sntrup761	2 051	1 694	7.5	3	400	35 465	88.66
NTT & CRT[1]			2 004	1 395	7.5	3	285	35 463	124.4
Schoolbook, HS, TW	$\mathcal{R}/q \cdot \mathcal{R}/3$	sntrup653	23 120	17 019	0	0	400	987	2.47
Schoolbook, HS, TW		sntrup761	27 693	19 555	0	0	400	1 149	2.87
NTT & CRT[1]			1 888	1 258	6.5	2	285	35 463	124.4
Schoolbook, HS[1]			27 594	20 078	0	0	289	1 522	5.27
Schoolbook, LA[1]			1 775	818	0	0	290	25 881	89.25
Karatsuba[2]			1 463	817	4	0	279	78 132	280.0
2x Karatsuba[3]			4 761	1 676	8.5	0	271	25 824	95.3
Schoolbook, [197]			38 798	21 768	2	0	312	762[4]	2.44[4]
Schoolbook, [186]			65 207	32 929	6	0	255	762[4]	2.99[4]
Schoolbook, TW		sntrup857	32 435	22 775	0	0	400	1 293	3.23
Schoolbook, TW	$\mathcal{R}/3 \cdot \mathcal{R}/3$	sntrup653	3 038	2 662	0	0	400	986	2.47
Schoolbook, TW		sntrup761	4 264	3 112	0	0	400	1 148	2.87
Schoolbook, TW		sntrup857	4 804	3 520	0	0	400	1 292	3.23

[1] We first presented this implementation in [17]. [2] We first presented this implementation in [10]. [3] Extension of our work in [10] by us, available at [12]. [4] Does not include the cycles to transfer the input and output.

Setting the bit-width to the proper value saves FF in the accumulator array, and LUT in the adders of the MAC units. The improved latency of the new schoolbook multiplier comes on the one hand from the higher clock frequency, and on the other hand from the ability to both load and output two coefficients per clock cycle, which reduces the total clock cycles. We can also see in Table 7.2 that the $\mathcal{R}/3 \cdot \mathcal{R}/3$ multipliers have essentially the same cycle count and latency as the $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplier, for all parameter sets. As expected however, the area usage is significantly smaller because the MAC units are simpler (compare Algorithm 10 and 9) and the accumulator array is smaller. Like with the $\mathcal{R}/q$ multiplier, the cycle count, LUT and FF usage increase nearly proportionally to the polynomial degree of the larger parameter sets. The one exception here is the LUT usage for sntrup653 parameter set, which is noticeably lower than one would expect. The exact cause for this difference has not been established yet.

There are two polynomial multipliers for Streamlined NTRU Prime in the literature [186, 197], both for the sntrup761 parameter set. Both also use the parallel schoolbook algorithm with an LFSR, though [197] uses the term x-net multiplier. In addition, both do not include the cycles for transferring the input operands or the resulting output. If one coefficient is transferred per clock cycle, this would add another $2p = 1522$ clock cycles, tripling the total cycle count. We can see that our improved high-speed schoolbook multiplier from this work has a lower area consumption than both multipliers from

the literature. In particular, our LUT usage is half of that from [186] and our clock frequency is 1.56 faster. The multiplier from [197] is reported to be slightly faster than ours, but this is to be expected due to the omission of transfer time. If we likewise omit the transfer time from our own multiplier, we have a cycle count of 761, for a latency of 1.90 µs which is 1.28 times faster than the multiplier from [197] and 1.57 times faster than [186]. This is due to our higher maximum clock frequency. Furthermore, our schoolbook multiplies does not need any BRAM, while the designs from [197] and [186] require two and six BRAMs respectively.

As an additional comparison, we also resynthesize a selection of our polynomial multiplication modules from Table 7.2 as an ASIC, using the 45 nm Nangate open cell library. We list the GE area in Table 7.3. Because the cell library does not contain a memory macro to implement SRAM, we list the memory footprint separately. We target a clock frequency of 100 MHz for all modules. Optimizations to achieve the maximum clock frequency would be of little benefit, as due to the missing SRAM cells, the timing analysis is incomplete. In Table 7.3, we can see that the high-speed Schoolbook multiplier is again the most expensive in terms of GE, while also being the fastest. The schoolbook multiplier also has the benefit of not needing any SRAM. The difference between the $\mathcal{R}/q \cdot \mathcal{R}/q$ and $\mathcal{R}/q \cdot \mathcal{R}/3$ NTT multiplier is also more pronounced, with the $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT multiplier needing 63% more GE, as well as 67% more SRAM. This increase is due to the additional multiplier and memory needed for the third CRT prime. While in the FPGA this increase in somewhat hidden in the single additional DSP, the full cost of such a large multiplier is immediately visible in an ASIC. The advantage of the low-area schoolbook multiplier over the NTT multiplier is also more pronounced: The schoolbook multiplier requires a third less GE and only 11% of the SRAM, while still having a lower latency. The Karatsuba multiplier also falls behind: while its GE is slightly lower that the low-area schoolbook multiplier, the Karatsuba multiplier requires over four times the SRAM and has over three times the cycle count. We can thus conclude that our schoolbook multipliers are also the best choice for an ASIC.

The advantage of the schoolbook polynomial multiplier over the Karatsuba and NTT multiplier has its origin in the fact that the schoolbook multiplier can better exploit the structure of the $\mathcal{R}/3$ polynomial. The $\mathcal{R}/3$ polynomial in Streamlined NTRU Prime has particularity small coefficient magnitude in comparison to other lattice schemes, with a range of only $[-1, 1]$. Other schemes generally have a range of $[-2, 2]$ to $[-5, 5]$ [32, 198]. The small range in Streamlined NTRU Prime allows the schoolbook multiplier to be particularly efficient, as it can directly be optimized to exploit the small range. Meanwhile, asymptotically faster algorithms such as Karatsuba or the NTT multiplier cannot exploit the $\mathcal{R}/3$ structure to the same degree. In the case of Karatsuba, the summation of the partial polynomials removes the small coefficients as the recursion depth increases, while the frequency transform and the multiplication inside the butterfly units of the NTT remove it entirely. The only benefit the NTT can use is only needing two CRT primes instead of three, as the maximum output range is smaller due to the

Table 7.3.: A comparison of ASIC area results in GE for our different multiplication algorithms for Streamlined NTRU Prime, using the 45nm Nangate open cell library [86]. The area does not include SRAM cells, which are listed separately.

Design	Op	Parameter	kGE	SRAM [kb]	Freq. [MHz]	Cycles
NTT & CRT [1]	$\mathcal{R}/q \cdot \mathcal{R}/q$		23.8	252	100	35 463
Schoolbook, TW			291.2	0	100	1 149
NTT & CRT [1]			15.2	168	100	35 463
Schoolbook, HS [1]	$\mathcal{R}/q \cdot \mathcal{R}/3$	sntrup761	295.2	0	100	1 522
Schoolbook, LA [1]			9.7	19.5	100	25 881
Karatsuba [2]			8.8	88	100	78 132
Schoolbook, TW	$\mathcal{R}/3 \cdot \mathcal{R}/3$		37.8	0	100	1 148

[1] We first presented this implementation in [17]. [2] We first presented this implementation in [10].

$\mathcal{R}/3$ polynomial. These factors lead to the schoolbook multiplier outperforming the asymptotically better algorithms. This of course only works for the $\mathcal{R}/q \cdot \mathcal{R}/3$ and $\mathcal{R}/3 \cdot \mathcal{R}/3$ multiplication. If one would try and implement the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier with the schoolbook algorithm, the multiplier would have a massive area consumption and be rather inefficient, as each MAC unit would need a full multiplier circuit. This allows the $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT multiplier to pull ahead, where the better asymptotic complexity leads to a more efficient design.

A further benefit of the schoolbook multiplier is the very simple memory access pattern: Both input polynomials are accessed in a purely sequential manner, as well as only in a single pass. In comparison, Karatsuba or the NTT multiplier have a much more complex access pattern. This simplicity allows us to store the input polynomials in simpler and smaller FIFOs, rather than fully addressable RAM. This in turn allows us to save BRAM or SRAM respectively in comparison to other multiplication algorithms.

A final distinguishing feature of the schoolbook multiplier is the high flexibility: The number of MAC units can be easily configured at design time, as can the coefficient and polynomial modules. As mentioned above, the memory access patterns and circuit layout are also simple. This allows the schoolbook multiplier to be quickly and easily adapted to almost any scenario to suit performance or resource constraints. In contrast, the NTT multiplier can require heavy modifications and tuning, especially if the target polynomial ring is not NTT friendly. This can be seen in the number of tricks needed to implement the NTT multiplier in Section 4.4 and 6.4. However, while the schoolbook multiplier is flexible at design time, the multiplier loses this flexibility after synthesis: In the FPGA or ASIC, the multiplier only supports a single scheme and parameter set. Although this also applies to our NTT multiplier, works in the literature have shown how so-called

unified designs can support multiple schemes and parameter sets simultaneously, using the same NTT multiplier [199–201]. Implementing something similar with a schoolbook multiplier would be difficult, as the different ranges of the small polynomials, the different coefficient moduli and polynomial moduli would lead to significant overheads.

7.1.3. Encoder and Decoder

Table 7.4 show a comparison of our new en- and decoder compared with our en- and decoder from [10] and [17]. The en- and decoder from [17] have either the same or lower resource consumption compared to those from [10], while at the same time significantly reducing the cycle counts, as well as increasing the max clock frequency. Our new en- and decoder do need more LUT and FF in comparison to those from [17]. In particular, our new encoder requires twice as many FF and 64% more LUT. However, the encoding latency also improves by a factor of four. In addition, the BRAM or DSP usage does not increase, and the absolute increase of LUT and FF is small. We can also see in Table 7.4 that the area change is minimal when switching parameter sets. The total cycle count increases linearly with the increasing polynomial degree of the larger parameter sets. A final point to consider is that the all the necessary multiplications in both encoding and decoding are all with different constants (e.g., a total of 42 for the decoder, as explained in Section 6.8). These are mapped to DSP by the synthesis tool but can also be performed fairly efficiently in LUT [202, 203]. This can be of use in cases where DSP are not available.

Table 7.4.: A comparison of our different encoding and decoding algorithms for Streamlined NTRU Prime. All cycle counts are for the $\mathcal{R}/q$ encode & decode of public keys. The target FPGA is a Xilinx Zynq Ultrascale+. Entries marked "TW" are new results from this thesis.

Design	Parameter	LUT	FF	BRAM	DSP	Freq. [MHz]	Cycles	Time [μs]
Encode, TW	sntrup653	322	312	0.5	1	400	677	1.69
Encode, TW		331	312	0.5	1	400	784	1.96
Encode[1]	sntrup761	201	154	0.5	1	290	2 297	7.92
Encode[2]		215	131	0.5	1	279	5 348	19.2
Encode, TW	sntrup857	313	312	0.5	1	400	880	2.20
Decode, TW	sntrup653	463	375	1	4	400	1 332	3.33
Decode, TW		447	377	1	4	400	1 549	3.87
Decode[1]	sntrup761	334	273	1	4	290	1 550	5.34
Decode[2]		676	571	2	5	279	7 380	26.4
Decode, TW	sntrup857	498	382	1	4	400	1 738	4.35

[1] We first presented this implementation in [17]. [2] We first presented this implementation in [10].

7.1.4. Polynomial Inversion

Table 7.5 shows a comparison of different inversion modules for Streamlined NTRU Prime and NTRU-HPS. All implement the extended GCD algorithm [87]. Our $\mathcal{R}/q$ inversion from [17] improves on our $\mathcal{R}/q$ inversion from [10], as due to a loop unroll factor of two, we gain a nearly two-times speedup. At the same, the amount of LUT and FF is reduced, and the DSP count remains the same. This is due to our improved modular reduction algorithm of [17]. Increasing the unroll factor to four gives another speedup of almost two, but also further increases DSP, FF and LUT consumption. In addition, distributed RAM is used instead of BRAM, as the BRAMs no longer have enough IO bandwidth. The increase of the number of pipeline stages in the $\mathcal{R}/q$ inversion, which is required for the higher clock frequency for the improved high-speed design, noticeably increases the number of FF by almost a factor of two. It also increases the total number of cycles, as the GCD algorithm has to flush the pipeline at certain points. Nevertheless, the increased clock frequency significantly reduces the total latency.

For the inversion in $\mathcal{R}/3$, it is clearly visible in Table 7.5 that increasing the loop unroll factor only leads to a comparatively small increase in resource consumption, in exchange for a significant increase in performance. While the inversion from [88] (used for the scheme NTRU-HPS) is over an order of magnitude faster when compared to this work or that of [17], it also uses significantly more LUT and FF than our designs, even when we use an unroll factor of 32. However, since the $\mathcal{R}/3$ inversion is not the bottleneck during key generation, further increasing the speed of our $\mathcal{R}/3$ inversion is unnecessary. The increase in clock frequency in this work has also not led to a meaningful increase in area usage compared to our inversion from [17]. For the sntrup857, we only use an unroll factor of 4 for the $\mathcal{R}/3$ inversion because we do not use batch inversion for this parameter set. This in turn gives us more time for the $\mathcal{R}/3$ inversion. In addition, we can see that for both the $\mathcal{R}/3$ and $\mathcal{R}/q$ inversion, the specific parameter set has only a minor influence in the area usage and maximum clock frequency. The only difference is a change in total clock cycles.

7.1.5. Modular Reduction

Table 7.6 shows a comparison of our different modular reduction modules. The DSP-free reduction from this work and [17] significantly outperforms our Barrett reduction from [10], for both modulo $q = 4591$ and modulo 3. For $q = 4591$, the design uses less LUT, no DSP and runs at a much higher clock frequency. The additional pipeline stages required to run the design at 400 Mhz do increase the amount of FF compared to [10] and [17], however, the absolute increase is still small. The reduction modules for the other Streamlined NTRU Prime q's have a similar area footprint. The area usage is slightly smaller for the CRT primes used in the NTT multiplier because the input ranges are smaller, which saves both LUT and FF. For modulo 3, our design uses more LUT and FF than [10], but the increase is insignificant compared to the increased clock frequency and the fact that no DSP are needed.

Table 7.5.: A comparison of different inversion modules for Streamlined NTRU Prime and NTRU-HPS. The target FPGA is a Xilinx Zynq Ultrascale+. The upper rows are for inversion of polynomials in $\mathcal{R}/q$, the lower in $\mathcal{R}/3$. All use the extended GCD algorithm [87]. Entries marked "TW" are new results from this thesis.

Design	unroll factor	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
$\mathcal{R}/q$, $p = 653$ TW	4	3 502	2 027	0	18	400	233 726	584.3
$\mathcal{R}/q$, $p = 761$ TW	4	3 497	2 009	0	18	400	313 479	783.7
$\mathcal{R}/q$, $p = 761$[1]	4	3 076	1 099	0	19	285	311 918	1 094
$\mathcal{R}/q$, $p = 761$[1]	2	1 122	593	2	11	285	600 906	2 108
$\mathcal{R}/q$, $p = 761$[2]	1	1 642	726	2	11	279	1 168 960	4 190
$\mathcal{R}/q$, $p = 857$ TW	4	3 410	1 728	0	18	400	394 128	985.3
$\mathcal{R}/3$, $p = 653$ TW	32	1 157	487	0	0	400	36 568	91.42
$\mathcal{R}/3$, $p = 761$ TW	32	1 169	479	0	0	400	47 167	117.9
$\mathcal{R}/3$, $p = 761$[1]	32	1 040	517	0	0	285	47 166	165.5
$\mathcal{R}/3$, $p = 761$[1]	2	607	211	0	0	285	590 158	2 071
$\mathcal{R}/3$, $p = 761$[2]	1	518	216	0	0	271	1 168 899	4 101
$\mathcal{R}/3$, $p = 821$ [88]	-	8 534	5 479	0	0	250	1 846	7.38
$\mathcal{R}/3$, $p = 857$ TW	4	513	223	0	0	400	380 298	950.7

[1] We first presented this implementation in [17]. [2] We first presented this implementation in [10].

7.1.6. Hash Module

Table 7.7 shows a comparison of different SHA-512 hash modules. Between [10], [17] and this work, we can see a steady improvement in the performance, in both a reduction in cycle count for a single SHA-512 round, as well as in an improved clock frequency. Our design from [17] has the lowest area cost, as the additional FIFO memory used in this work does increase the LUT count substantially. The benefit of the FIFO memory is not directly visible in Table 7.7, but can be seen in the operation scheduling diagrams in Figure 7.7 and 7.8 in Section 7.5. The design from this work also outperforms the SHA-512 designs from [204] and [205], with a much higher frequency and a lower total latency, despite our cycle count being higher. However, this is also to be expected, as we use a newer FPGA.

Table 7.7 also includes the Keccak RNG (see Section 6.12). The area numbers include the buffers needed to store the random 32 bit words prior to usage. While the LUT consumption is higher than our SHA-512 module, we can also see that Keccak has higher performance in hardware, due to the much lower cycle count per input block. Nevertheless, the comparatively high LUT consumption does show that not including an RNG can distort the total area. We urge future work to always specify whether an RNG is included or not, in order to properly compare implementations.

Table 7.6.: A comparison of our different modular reduction modules for Streamlined NTRU Prime and other PQC algorithms. The target FPGA is a Xilinx Zynq Ultrascale+. Entries marked "TW" are new results from this thesis.

Design	LUT	FF	BRAM	DSP	Freq. [MHz]
Mod $q = 4591$, TW	160	146	0	0	400
Mod $q = 4591$ [1]	157	99	0	0	290
Mod $q = 4591$ [2]	304	107	0	1	279
Mod $q = 4621$, TW	157	150	0	0	400
Mod $q = 5167$, TW	158	124	0	0	400
Mod $q = 7861$, TW	78	72	0	0	400
Mod $q = 12289$, TW	105	69	0	0	400
Mod $q = 15361$, TW	83	76	0	0	400
Mod 3, TW	26	69	0	0	400
Mod 3 [2]	23	19	0	1	279

[1] We first presented this implementation in [17].
[2] We first presented this implementation in [10].

Table 7.7.: A comparison of different hash algorithms for Streamlined NTRU Prime and other PQC algorithms. The cycle count is always for a single input block. The target FPGA is a Xilinx Zynq Ultrascale+, except for [204], which is a Xilinx Virtex-4 and [205], which is a Xilinx Virtex-6 FPGA. None of the designs consume DSP. Entries marked "TW" are new results from this thesis.

Design	Slices	LUT	FF	BRAM	Freq. [MHz]	Cycles	Time [ns]
SHA-512, TW	555	3 336	1 841	0	400	90	225
SHA-512 [1]	391	2 319	1 423	0	290	117	403
SHA-512 [2]	716	3 174	4 710	1	279	325	1 165
SHA-512 [204]	1 520	2 751	1 689	0	170.4	80	469
SHA-512 [205]	1 811	-	-	-	192.6	84	436
Keccak RNG, TW	611	4 410	1 826	0	400	24	60

[1] We first presented this implementation in [17].
[2] We first presented this implementation in [10].

7.2. Evaluation of the Full Designs

In this section, we will evaluate and compare the various Streamlined NTRU Prime implementations that were written as part of this thesis, including [10] and [17]. We will first evaluate the low-area designs, followed by the high-speed designs. As part of the analysis, we evaluate the impact of different batch sizes and loop unroll factors for inversion, and their influence on key generation speed and area usage.

7.2.1. Evaluation of the Low-Area Designs

In this section we will compare the low-area Streamlined NTRU Prime implementations that were developed as part of this thesis. They have been previously published in [10] and [17]. Both employ the parameter set sntrup761. The design from [17] targets both the Zynq Ultrascale+ and the Artix-7, whereas [10] targets only the Zynq Ultrascale+. The benchmark numbers of individual operations of our implementation for the Zynq Ultrascale+ and the Artix-7 are listed in Table 7.8 and 7.9 respectively. In addition, benchmark numbers for a full, merged design are given in Table 7.10. For our design of [10], we use the newest numbers from our GitHub repository [12], which have slightly lower cycle counts and resource utilization. Note that both designs do not contain a random number generator. This mirrors the reference design of Streamlined NTRU Prime [9] and allows us to directly use the inputs of the known-answer-test to verify the correctness of our design. We can see that while our design from [17] does require slightly more LUTs (at most 31% more) than our lightweight implementation from [10], the implementation from [17] is significantly faster, with a 2.05, 4.08 and 3.04 speedup respectively for key generation, encapsulation and decapsulation. In addition, our lightweight implementation from [17] uses less DSP and BRAM than [10], both for individual operations as well as in the merged design.

Table 7.8.: Our low-area designs implemented on a Xilinx Zynq Ultrascale+ FPGA.

Design	Parameter	Module	Slices	LUT	FF	BRAM	DSP	Freq. [MHz]	Cycles	Time [μs]
SNTRUP, LA [17]	sntrup761	Key Gen	1 232	7 216	3 726	5.5	12	285	629 367	2 208
		Encap	1 074	6 030	3 211	4.5	7	290	29 245	100.8
		Decap	1 051	6 016	3 194	3	7	283	85 628	302.6
SNTRUP [10, 12]	sntrup761	Key Gen	1 068	5 935	4 144	11.5	12	271	1 289 959	4 748
		Encap	844	4 570	2 843	7.5	8	271	119 250	439
		Decap	902	5 117	2 958	7	8	271	260 307	958

7.2.2. Evaluation of the High-Speed Designs

In this section we will compare the high-speed Streamlined NTRU Prime implementations that were developed as part of this thesis. This includes the previously published work in [17], as well as the improvements presented in this thesis. [17] supports the parameter set sntrup761, while our newly presented design also supports sntrup653 and sntrup857,

Table 7.9.: Our low-area designs implemented on a Xilinx Artix-7 FPGA. As to be expected of the lower-end platform, the design uses more LUT and has a lower maximum clock frequency when compared to the Zynq Ultrascale+.

Design	Parameter	Module	Slices	LUT	FF	BRAM	DSP	Freq. [MHz]	Cycles	Time [μs]
SNTRUP, LA [17]	sntrup761	Key Gen	2 376	7 579	3 824	5.5	12	159	629 367	3 958 μs
		Encap	1 945	6 379	3 069	4.5	6	147	29 245	198.9 μs
		Decap	1 842	6 279	3 086	3	7	131	85 628	653.6 μs

Table 7.10.: Full implementation all our low-area designs, with all operations merged.

Design	Parameter	Platform	Slices	LUT	FF	BRAM	DSP	Freq. [MHz]
SNTRUP, LA [17]	sntrup761	Zynq Ultrascale+	1 539	9 154	4 423	8.5	18	285
		Artix-7	2 968	9 574	4 399	8.5	18	128
SNTRUP [10, 12]	sntrup761	Zynq Ultrascale+	1 367	7 807	4 144	11.5	19	271

though batch key generation is not available for the sntrup857 parameter set. The benchmark numbers of individual operations of our implementations for the Zynq Ultrascale+ and the Artix-7 are listed in Table 7.11 and 7.12 respectively. In addition, benchmark numbers for a full, merged design are given in Table 7.13. Note that the design in [17] does not contain a random number generator, while our improved design contains the Keccak-based RNG (see Section 6.12). The inclusion of the Keccak-based RNG does add a noticeable overhead, as it consumes up to 13.2 % of the LUT. In total, our design from this work does consume more LUT and FF across all operations, though it does need less BRAM and DSP, while also improving both the cycle count and the maximum clock frequency on both FPGA platforms. This leads to a significantly improved latency for all operations.

Both high-speed designs use batch inversion when available. Table 7.14 compares the cycle counts for different batch sizes for the parameter set sntrup761. Larger batches increase the total number of cycles to complete the batch, but dramatically decrease the amortized cycles per key. This is a major downside for our implementation of [17], which has an initial very large latency as the whole batch is calculated. However, in the improved high-speed design of this work, the total number of cycles until the *first* key is ready is much lower: Only 449 588 cycles for a batch size of 21, compared to a non-batch cycle count of 317 542, an increase of only 41.5 %. This allows applications to already use the first key of the batch, negating the drawback of having to wait until the full batch is complete. In general, a batch size of 21 (together with an unroll factor of four) appears optimal for the parameter set sntrup761, as the speedup from increasing the batch size from 21 to 42 is relatively low.

Table 7.11.: Our high-speed designs implemented on a Xilinx Zynq Ultrascale+ FPGA. Encapsulation and key generation assume short polynomials have been pregenerated. The key generation cycle counts assume a batch size of 24 for sntrup653 and 21 for sntrup761 and list the amortized per-key cycles. Key generation for sntrup857 does not use batch inversion.

Design	Parameter	Module	Slices	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
SNTRUP, HS, TW	sntrup653	Key Gen	6 409	39 767	25 805	23.5	22	400	52 719	131.8
		Encap	5 182	33 387	21 957	6	5	400	2 252	5.63
		Decap	5 452	32 559	22 852	2.5	5	400	3 727	9.32
SNTRUP, HS, TW	sntrup761	Key Gen	7 297	43 547	28 465	23.5	22	400	59 320	148.3
		Encap	6 012	37 314	24 570	6	5	400	2 571	6.43
		Decap	6 547	37 871	26 070	2.5	5	400	4 316	10.8
SNTRUP, HS, [17]	sntrup761	Key Gen	6 038	37 813	25 368	33	23	285	64 026	224.7
		Encap	5 381	31 996	22 425	4.5	6	289	5 007	17.3
		Decap	5 432	32 301	22 724	3.5	9	285	10 989	38.6
SNTRUP, HS, TW	sntrup857	Key Gen	7 357	47 577	29 173	6	19	400	398 708	996.8
		Encap	6 925	44 209	27 831	6	5	400	2 864	7.16
		Decap	7 498	45 392	29 912	2.5	5	400	4 842	12.1

Table 7.12.: Our high-speed designs implemented on a Xilinx Artix-7 FPGA. As to be expected of the lower-end platform, the design uses more LUT and has a lower maximum clock frequency when compared to the Zynq Ultrascale+.

Design	Parameter	Module	Slices	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
SNTRUP, HS, TW	sntrup761	Key Gen	14 902	48 464	28 339	23.5	22	193.2	59 320	307
		Encap	13 194	44 157	24 724	3	5	186.1	2 571	13.8
		Decap	13 167	43 446	26 123	2.5	5	187.5	4 316	23.0
SNTRUP, HS, [17]	sntrup761	Key Gen	10 827	39 200	25 536	33.5	23	143	64 026	447.7
		Encap	11 218	40 879	22 382	4.5	6	144	5 007	34.8
		Decap	10 169	36 789	22 700	3.5	9	137	10 989	80.2

At the same time, our key generation is highly tweakable, as both the batch size and the unroll factor of the $\mathcal{R}/q$ inversion can be easily configured, allowing our design to make optimal use of available DSP and BRAM resources. To demonstrate this, we also synthesize our improved high-speed design with an inversion unroll factor of one (i.e., no loop unrolling) but a batch size of 84, as well as for an unroll factor of 32 and a batch size of one (i.e., no batch inversion). We use the parameter set sntrup761 in both cases. The results can be found in Table 7.15, where we also compare our improved high-speed design. We can see that all implementations have a very similar latency. However, the non-batch implementation has an extremely high DSP usage of 131, as well as the highest LUT and FF consumption. It does however have a much lower BRAM consumption, which is particularly high for the implementation using a batch size of 84. In return, the batch-84 design does have the lowest DSP usage. Using a balanced approach (with a batch size of 21, as well as a small unrolling of the inversion itself) gives us a fast key generation speed while also not consuming too many BRAM or DSP. For this reason, we conclude that a balanced approach is generally the optimal choice, as long as there

Table 7.13.: Full implementation of all our high-speed designs, with all operations merged.

Design	Parameter	Platform	Slices	LUT	FF	BR AM	DSP	Freq. [MHz]
SNTRUP, HS, TW	sntrup653	Zynq Ultrascale+	7 724	46 135	30 025	25.5	26	400
		Artix-7	15 758	51 518	29 821	25.5	26	193.9
SNTRUP, HS, TW	sntrup761	Zynq Ultrascale+	8 770	52 106	33 130	25.5	26	400
		Artix-7	17 397	57 923	33 156	25.5	26	184.8
SNTRUP, HS [17]	sntrup761	Zynq Ultrascale+	7 051	40 060	26 384	36.5	31	285
		Artix-7	11 745	41 428	26 381	36.5	31	140
SNTRUP, HS, TW	sntrup857	Zynq Ultrascale+	9 018	55 837	34 427	8	23	400
		Artix-7	18 296	61 187	34 380	8	23	181.8

Table 7.14.: The effect of the different batch sizes on the speed of key generation, with an unroll factor of four for the $\mathcal{R}/q$ inversion and the parameter set sntrup761. The clock frequency and other FPGA resources are only minimally affected by increasing the batch size.

Batch Size	Total cycles	Amortized cycles	First key ready	BRAM
1	317 542	317 542	317 542	6
5	505 016	101 004	374 164	16
21	1 245 704	59 320	449 588	23.5
42	2 216 202	52 767	546 928	33

are no additional constraints on BRAM or DSP usage. A side remark is that using both a high unroll factor and a large batch size quickly becomes pointless, as the additional multiplications needed for batch inversion dominate the runtime.

Table 7.15.: A comparison of key generation of our improved high-speed design with different batch sizes and loop unroll factors for the inversion.

Design	Batch Size	Unroll Factor	Slices	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
sntrup761	21	4	7 297	43 547	28 465	23.5	22	400	59 320	148.3
	1	32	8 981	53 485	36 235	6	131	400	63 553	158.9
	84	1	7 026	41 357	27 574	54	10	400	59 870	149.7

A interesting detail for future work is that the memory used to store the intermediate polynomials does not strictly have to be in BRAM: If we look at Algorithm 4, we can see that the polynomials a_i and f_i are not accessed for considerable periods of time. It would thus be feasible to move these polynomials to cheaper (but slower) memory such as Dynamic Random-Access Memory (DRAM) while they are not needed and copy them back to BRAM before they are used. The n a_i polynomials are the main source

of BRAM usage for a batch size of n (e.g., 26 BRAM for $n = 84$). Moving the smaller $\mathcal{R}/3$ polynomials f_i, g_i and g_i^{-1} to DRAM would save another 12 BRAM. Storing the four polynomials in DRAM would allow us to use larger batch sizes while reducing the BRAM usage. We could then in turn either increase the key generation speed further or reduce the unroll factor, saving DSPs. All-in-all, our results show batch inversion is feasibly in many scenarios, and its high performance significantly accelerates the key generation of Streamlined NTRU Prime.

7.2.3. Side-channel Security of our Designs

For our low-area and high-speed designs, our focus was on performance and efficiency, and not advanced side-channel security. As a result, they are not protected against more powerful SCAs. Nevertheless, all of our designs are fully constant-time with regards to secret input and are thus immune to timing SCAs. The only operation that has any timing variations is the $\mathcal{R}/3$ inversion, which in rare cases can find that the input polynomial in not invertible. In these cases, rejection sampling is used, and a new polynomial is sampled and inverted. This is not vulnerable to SCAs. A further note is that the radix sorting used in the generation of short polynomials does include secret-dependent memory indexing. However, because the BRAMs on FPGAs have no cache, this also does not expose a side channel.

7.3. Comparison with Other Designs

We will now compare our Streamlined NTRU Prime implementations with those from the literature [206], as well as with NTRU [88, 207], Kyber [88] and Saber [88]. NTRU, Kyber and Saber are all round 3 finalist KEM algorithms in the NIST standardization process and like NTRU Prime are based on structured lattices [32, 198, 208]. NTRU has two variants, NTRU-HPS and NTRU-HRSS. We include both in our comparisons. In general, we will focus our comparison on that with NTRU, due to the similarities between Streamlined NTRU Prime and NTRU. The comparison can be found in Table 7.16. The area usage for the improved Streamlined NTRU Prime design from this thesis includes the Keccak RNG (see Section 6.12), removing the RNG would reduce the LUT count by approximately 4 400 and the FF count by 1 800 for encapsulation and key generation.

Table 7.16.: A comparison of different Streamlined NTRU Prime implementations, as well as a selection of NTRU, Kyber and Saber implementations. Our designs are marked with TW, and HS denotes a high-speed version, and LA a low-area version. All entries are implemented on a Xilinx Zynq Ultrascale+ FPGA.

Module	Design	Slices	LUT	FF	BRAM	DSP	Freq. [MHz]	Cycles	Time [μs]
				Security Level 1					
Keygen	NTRU-HPS667 [88]	7 698	41 047	39 037	6	45	250	48 179	192.7
	NTRU-HRSS701 [88]	9 357	49 001	39 957	2.5	45	300	51 812	172.2
	sntrup653, HS, TW[4]	6 409	39 767	25 805	23.5	22	400	52 719	131.8
	Kyber L1 [88]	-	9 504	8 957	4.5	4	450	2 200	4.9
	Saber L1 [88]	3 844	23 557	14 190	1.5	0	370	1 607	4.3
Encap	NTRU-HPS667 [88]	4 638	26 325	17 568	5	0	250	3 687	14.7
	NTRU-HPS667 [207]	4 509	24 664	13 902	8.5	0	400	5 928	14.8
	NTRU-HRSS701 [88]	6 652	31 494	25 120	2.5	0	300	2 219	7.4
	NTRU-HRSS701 [207]	4 699	28 396	15 894	9	0	400	2 879	7.2
	sntrup653, HS, TW[4]	5 182	33 387	21 957	6	5	400	2 252	5.63
	sntrup653, [206][3]	9 110	62 797	33 531	9	0	278	-	52.3
	Kyber L1 [88]	-	9 504	8 957	4.5	4	450	3 200	7.2
	Saber L1 [88]	3 984	24 199	14 457	1.5	0	370	2 153	5.8
Decap	NTRU-HPS667 [88]	5 217	29 935	19 511	2.5	45	300	7 522	25.1
	NTRU-HPS667 [207]	5 426	23 689	21 286	6	667	350	6 163	17.6
	NTRU-HRSS701 [88]	8 032	37 702	34 441	2.5	45	300	8 826	29.4
	NTRU-HRSS701 [207]	6 257	27 790	24 979	6	701	350	7 606	21.7
	sntrup653, HS, TW[4]	5 452	32 559	22 852	2.5	5	400	3 727	9.32
	sntrup653, [206][3]	9 110	62 797	33 531	9	0	278	-	52.3
	Kyber L1 [88]	-	9 504	8 957	4.5	4	450	4 500	10.0
	Saber L1 [88]	4 364	24 655	14 879	1.5	0	370	2 794	7.6

continued on next page

[1] We first presented this implementation in [17].
[2] We first presented this implementation in [10].
[3] Does not implement decoding.
[4] Improved high-speed design first presented in this thesis.

Table 7.16.: (continued)

Module	Design	Slices	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
				Security Level 2					
Keygen	sntrup761, HS, TW[4]	7 297	43 547	28 465	23.5	22	400	59 320	148.3
	sntrup761, HS, TW[1]	6 038	37 813	25 368	33	23	285	64 026	224.7
	sntrup761, LA, TW[1]	1 232	7 216	3 726	55	12	285	629 367	2 208
	sntrup761, TW[2]	1 068	5 935	4 144	11.5	12	271	1 289 959	4 748
Encap	sntrup761, HS, TW[4]	6 012	37 314	24 570	6	5	400	2 571	6.43
	sntrup761, HS, TW[1]	5 381	31 996	22 425	4.5	6	289	5 007	17.3
	sntrup761, LA, TW[1]	1 074	6 030	3 211	4.5	7	290	29 245	100.8
	sntrup761, TW[2]	844	4 570	2 843	7.5	8	271	119 250	439
	sntrup761, [206][3]	10 319	70 066	38 144	9	0	263	-	56.3
Decap	sntrup761, HS, TW[4]	6 547	37 871	26 070	2.5	5	400	4 316	10.8
	sntrup761, HS, TW[1]	5 432	32 301	22 724	3.5	9	285	10 989	38.6
	sntrup761, LA, TW[1]	1 051	6 016	3 194	3	7	283	85 628	302.6
	sntrup761, TW[2]	902	5 117	2 958	7	8	271	260 307	958
	sntrup761, [206][3]	10 319	70 066	38 144	9	0	263	-	53.3
				Security Level 3					
Keygen	NTRU-HPS821 [88]	10 127	50 347	44 281	6.5	45	250	67 157	268.6
	sntrup857, HS, TW[4]	7 357	47 577	29 173	6	19	400	398 708	996.8
	Kyber L3 [88]	-	10 590	10 458	6.5	6	450	2 600	5.9
	Saber L3 [88]	3 634	20 496	13 939	1.5	0	370	2 709	7.3
Encap	NTRU-HPS821 [88]	7 370	33 698	30 551	5.5	0	250	4 576	18.3
	NTRU-HPS821 [207]	4 978	29 637	16 634	9	0	400	7 181	17.9
	sntrup857, HS, TW[4]	6 925	44 209	27 831	6	5	400	2 864	7.16
	sntrup857, [206][3]	11 509	78 379	42 274	9	0	250	-	60.1
	Kyber L3 [88]	-	10 590	10 458	6.5	6	450	3 200	8.3
	Saber L3 [88]	3 321	21 069	14 074	1.5	0	370	3 735	10.1
Decap	NTRU-HPS821 [88]	7 785	38 642	33 003	2.5	45	300	10 211	34.0
	NTRU-HPS821 [207]	6 808	29 074	26 474	6	821	350	7 521	21.4
	sntrup857, HS, TW[4]	7 498	45 392	29 912	2.5	5	400	4 842	12.1
	sntrup857, [206][3]	11 509	78 379	42 274	9	0	250	-	57.3
	Kyber L3 [88]	-	10 590	10 458	6.5	6	450	4 500	10.9
	Saber L3 [88]	3 816	21 342	14 233	1.5	0	370	4 682	12.7

[1] We first presented this implementation in [17].
[2] We first presented this implementation in [10]. [3] Does not implement decoding.
[4] Improved high-speed design first presented in this thesis.

To our knowledge, there are no full Streamlined NTRU Prime implementations other than ours in the literature. However, Dang et *al.* [206] presented a hardware-software co-design of round 2 Streamlined NTRU Prime for the parameter sets sntrup653, sntrup761 and sntrup857. It only implements the encapsulation and decapsulation, and omits the key generation. In addition, the decoding is also not implemented in hardware. For a fairer comparison, we only compare the parts implemented in hardware. Because it also a high-speed design, we will compare it with the improved high-speed design from this work. Despite implementing less in hardware, the design uses significantly more FPGA resources, in particular LUT, across all parameter sets. At the same time, the execution time of [206] is at least 4.7 and 8.4 times longer for decapsulation and encapsulation respectively, compared to our high-speed designs.

When comparing our improved high-speed implementation of sntrup653 with that of NTRU-HPS667 from [88], we can see that our encapsulation does use more LUT, BRAM, DSP and FF. At the same time, we have a lower cycle count by a factor of 1.6, as well a much higher clock frequency. These two factors lead to a significantly faster execution time, which is better by a factor of 2.6. For decapsulation, our design uses slightly more LUT and FF, the same amount of BRAM but significantly fewer DSP. At the same time, our cycle count is lower by a factor of 2.01. This lower cycle count, together with the faster clock speed, leads to a faster execution by a factor of 2.69. For key generation, our design uses fewer LUT, FF, DSP, while having a slightly higher cycle count but a much faster clock speed, for a total latency improvement of 1.46. However, our design does use roughly four times as much BRAM due to the batch inversion. When we compare NTRU-HPS821 from [88] with our implementation of sntrup857, we see a similar picture: Our encapsulation uses a third more LUT, but less FF, with slightly more BRAM and DSP. At the same time our latency is again significantly lower, by a factor of 2.5. Likewise, our decapsulation uses more LUT, fewer FF, significantly fewer DSP while being much faster: Our design has a latency of just 35.5 % of the latency of [88]. However, our key generation for sntrup857 is massively slower compared to the key generation of NTRU-HPS821 from [88] due to the lack of batch inversion, though our area consumption is lower. In particularly, our design uses only 65.8 % of the FF and 42.2 % of the DSP.

We also compare our sntrup653 design with the NTRU-HRSS variant of NTRU from [88]. Our encapsulation uses 5.3 % more LUT, but 12.6 % less FF, while having a slightly higher cycle count. However, the faster clock frequency of our design allows us to have a lower overall latency by a factor of 1.31. For decapsulation, our design uses less of every FPGA resource, while also having both a faster clock frequency and a lower cycle count. This results in our decapsulation having a latency of just 31.7 % of that of NTRU-HRSS. The key generation comparison is similar, where our design uses less of every resource except BRAM. Our design does have a slightly high cycle count, but thanks to our higher frequency, we have a lower latency.

In addition, we compare our sntrup653 design with the recently published NTRU-HPS667 implementation from [207], though they did not implement key generation. Their encapsulation uses 26 % less LUT and 36 % less FF, while using slightly more BRAM. Their design has the same clock frequency as ours, but due to their much higher cycle count, is also slower by a factor of 2.56. For decapsulation, while their design uses 27 % less LUT and 8 % less FF, it also uses over 60 times the DSPs. At the same time, our design has a faster latency by a factor of 1.88.

A final detail we wish to highlight is that all of the DSPs of the encapsulation and encapsulation of our improved high-speed design are from the encoder and decoder. However, as mentioned in Section 7.1.3, these are all used for multiplications with constants. Thus, they could be implemented relatively efficiently in LUT if DSP are not

available, as constant multipliers are much cheaper than full multipliers [202, 203]. This also means that *no full multipliers* are required at all for encapsulation or decapsulation for Streamlined NTRU Prime. This is of course not the case for key generation, where we need full multipliers for the NTT and the divstep computation. In contrast, Kyber requires full multipliers for all three operations because Kyber specifies the use of an NTT [32, 88]. Likewise, NTRU also requires full multipliers for key generation and decapsulation, as it includes a "big×big" multiplication [88, 208]. Saber on the other hand can also exploit the "big×small" property of its polynomials and use a schoolbook polynomial multiplication algorithm that does not need any full multipliers [88, 198].

7.4. Comparing Core-SVP, Latency and Speed-Area Product

In addition to listing the numbers in Table 7.16, we also plot the Core-SVP security of the different schemes and parameter sets against the encapsulation and decapsulation latency in Figure 7.1 and 7.2. Comparing the Core-SVP security is helpful compared to just the NIST security levels, as there can be leeway within a single category [209,210], while also allowing us to better visualize how the designs compare across multiple parameter sets. Informally, the Core-SVP metric estimates in bits how difficult it would be to directly solve the SVP of the underlying lattice of the cryptosystem when running the BKZ algorithm [59]. Though not without criticism [9,51], Core-SVP has seen widespread use in comparing lattice KEMs [31,51,209]. For this comparison, we compare our improved high-speed design with NTRU-HPS, NTRU-HRSS, Saber and Kyber from [88][1].

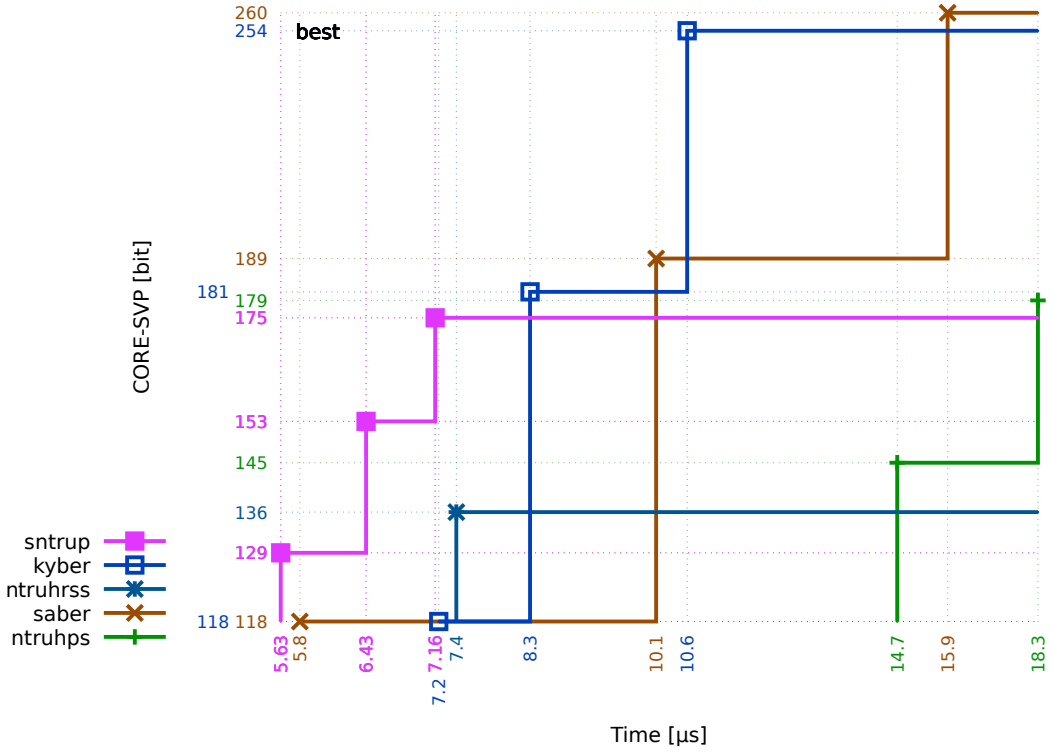


Figure 7.1.: A diagram comparing Core-SVP vs. encapsulation speeds of structured lattice KEMs. Each data point corresponds to a specific parameter set of the associated design. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

[1]The scripts to generate these plots are originally from [209], we express our thanks to Daniel J. Bernstein for providing them and allowing their reuse.

In Figure 7.1, we can see that our design outperforms all other KEMs for encapsulation, offering higher Core-SVP, faster speed, or both. It also significantly outperforms the NTRU-HPS variant of NTRU. We conjecture that the larger Streamlined NTRU Prime parameter sets, which we have not implemented, would also perform similarly well. For the decapsulation (Figure 7.2), our design is on par with Kyber and Saber, while significantly outperforming both NTRU variants.

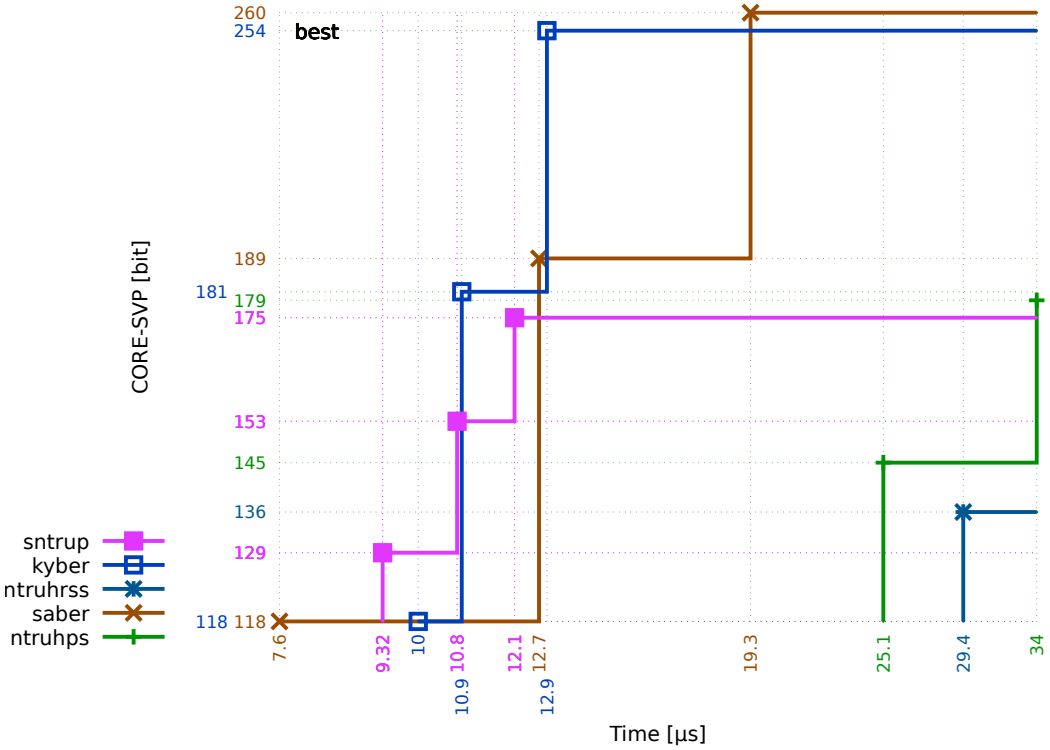


Figure 7.2.: A diagram comparing Core-SVP vs. decapsulation speeds of structured lattice KEMs. Each data point corresponds to a specific parameter set of the associated design. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

Figure 7.3 depicts Core-SVP security against the key generation latency. As in Table 7.16, we use a batch size of 24 and 21 for the parameter sets sntrup653 and sntrup761 respectively and use non-batch inversion for the sntrup857 parameter set. We can immediately see that Streamlined NTRU Prime and both NTRU variants fall behind Kyber and Saber significantly, as all three require a computationally expensive polynomial inversion during key generation. However, our Streamlined NTRU Prime implementation outperforms both NTRU variants for the parameter sets sntrup653 and sntrup761, thanks to the speed-up from the batch inversion. When batch inversion cannot be used as in

the case of sntrup857, then Streamlined NTRU Prime falls behind NTRU. We would like to stress here that there is no fundamental reason why one could not use batch inversion for sntrup857. Our inability of doing so is solely due to the limitations of our $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT multiplier. Using for example Good's trick with 5 $\mathrm{NTT}_{2^9}(\cdot)$ instead of 3 would allow us to perform $\mathcal{R}/q \cdot \mathcal{R}/q$ polynomial multiplication for sntrup857 as well. In fact, because $512 \cdot 5 = 2560 > 2554 = 1277 \cdot 2$, such an NTT multiplier would be sufficient for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication even for the sntrup1277 parameter set.

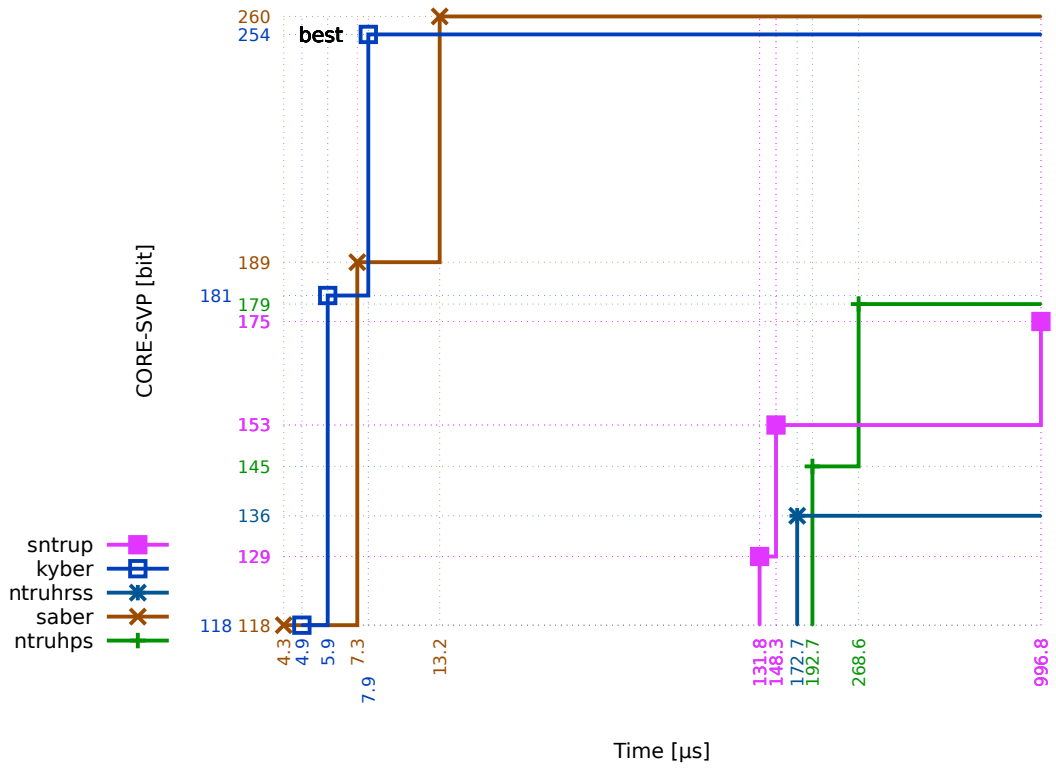


Figure 7.3.: A diagram comparing Core-SVP vs. key generation speeds of structured lattice KEMs. Each data point corresponds to a specific parameter set of the associated design. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

Figures 7.1, 7.2 and 7.3 only compare the operation latency vs. Core-SVP, ignoring area and FPGA resource consumption. For this, we need the time-area product of the different operations. The best way of creating the time-area product for FPGAs is debatable, as there are multiple factors that contribute to the area consumption: LUT, FF, BRAM, DSP, slices etc. In [88], the authors suggest using only the number of LUT because their number is the most limiting on FPGAs. We will be using this metric for our comparison as well. This method has the downside of ignoring the use of all other

FPGA components, such as the high BRAM usage of our batch inversion or the very high DSP usage of NTRU-HPS667 of [207]. Furthermore, components such as RAM and DSP multipliers can also skew the area comparison, as these components have a considerable area cost in an ASIC, whereas they are available in high numbers in an FPGA [211]. Circuits that implement full multiplications, such as needed in the NTT for Kyber or during our batch inversion, are also much more expensive in an ASIC than constant multipliers [202, 203], as needed for example in our decoder. However in an FPGA, both appear identically in the area report as a DSP. In addition, a single LUT can represent a hardware function of different complexity: For example, a $\text{LUT}_{6,1}$ can implement a 6 input NAND gate, or a 64 bit SRAM memory, both of which have vastly different area usage in an ASIC. As a result, these time-area products should be treated with some caution, especially when deducing area costs for eventual ASIC implementations. If of course FPGAs are indeed the target platform, then these time-area diagrams can be used directly.

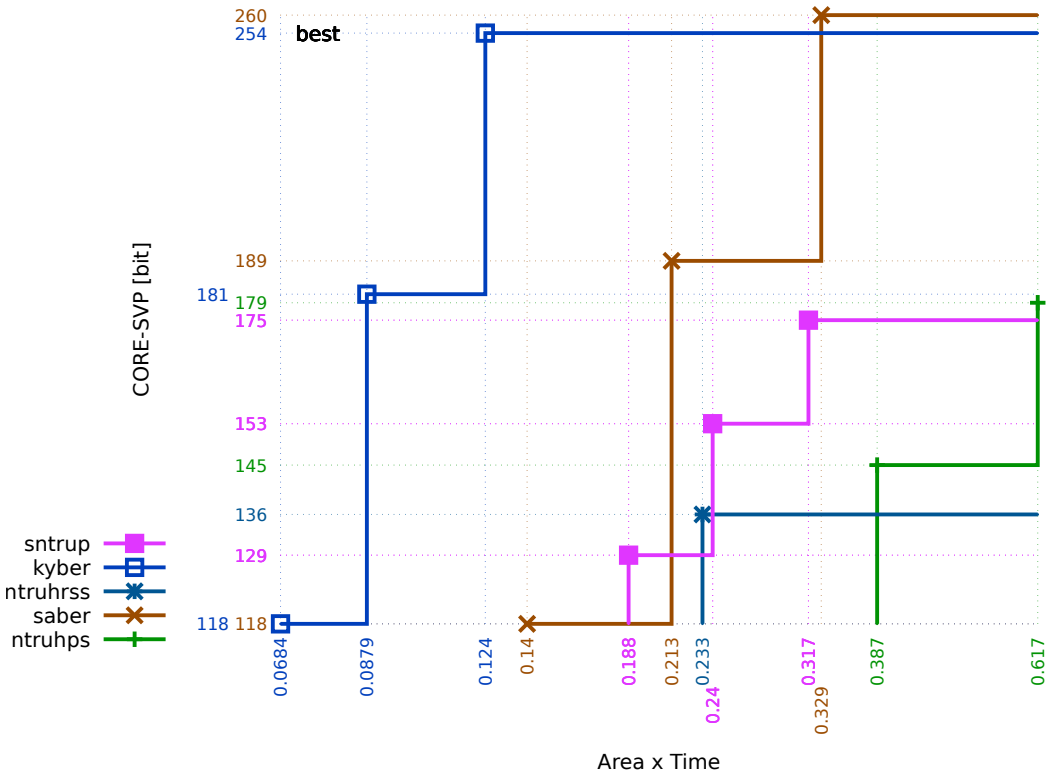


Figure 7.4.: A diagram comparing Core-SVP vs. encapsulation time-area product of structured lattice KEMs. Time-area product is calculated by the number of LUT times the latency. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

Figures 7.4, 7.5 and 7.6 show the time-area product for all three main operations of structured lattice KEMs. For encapsulation, we can see that our design is slightly behind Saber, while outperforming NTRU-HPS and being on-par with NTRU-HRSS. For decapsulation, Streamlined NTRU Prime falls further behind Saber, though also now clearly outperforming both NTRU variants. During key generation, we see a similar picture as in Figure 7.3, with Streamlined NTRU Prime being significantly behind both Saber and Kyber, while also being ahead of the NTRU variants as long as batch inversion can be used. For all three operations, Kyber is noticeably ahead of all other schemes.

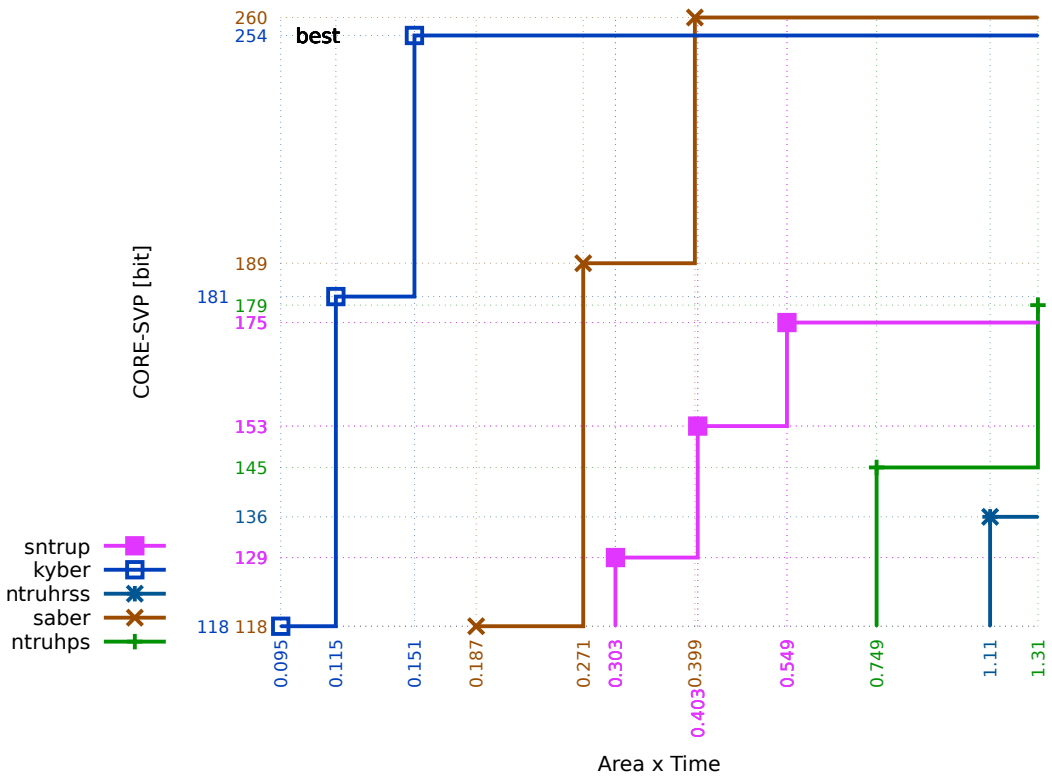


Figure 7.5.: A diagram comparing Core-SVP vs. decapsulation and time-area products of structured lattice KEMs. Time-area product is calculated by the number of LUT times the latency. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

However, as mentioned previously this comparison on FPGAs is somewhat inaccurate, as we only take the number LUT into account when computing the time-area product, while BRAM and DSP can have a considerable cost attached to them. It would thus be of interest to repeat this comparison while using ASIC designs of all schemes using the same cell library. Using the same cell library would be ideal in order to make the area result directly comparable, in a similar fashion as using the same FPGA is needed for a proper comparison. However, we could also use the GE of a design because this number

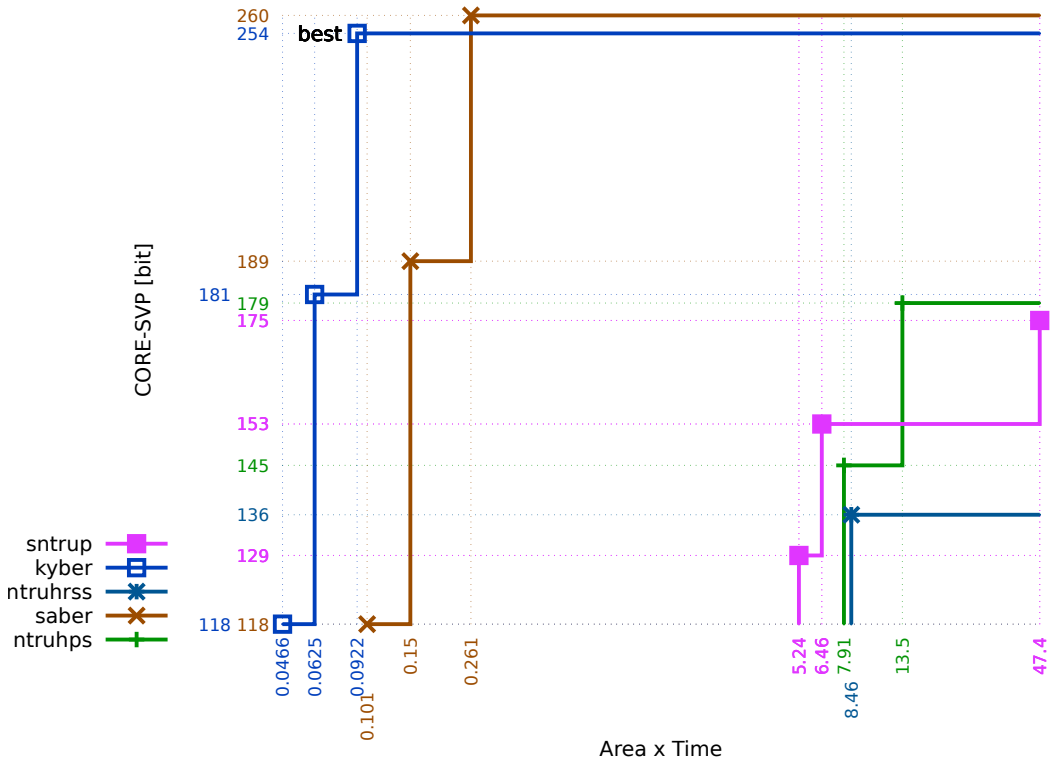


Figure 7.6.: A diagram comparing Core-SVP vs. key generation time-area products of structured lattice KEMs. Time-area product is calculated by the number of LUT times the latency. The numbers for sntrup are for the improved high-speed design from this work, all others are from [88]. All axes are log scale.

is independent of the cell library. The time-area product would then be calculated as the latency multiplied with the GE of a design. Looking beyond the comparison, a full ASIC design of Streamlined NTRU Prime would in generally be of interest, and we hope we will be able to present one at some point. For now, we leave this for future work.

7.5. Scheduling Diagrams and Future Improvements

In this section we have deeper look in the instruction scheduling of our high-speed implementation for the parameter set sntrup761. As explained in Section 6.12, the improved scheduling is one of the reasons for the improved performance over our high-speed design of [17]. The instruction scheduling diagrams for the sntrup761 parameter set can be found for encapsulation (Figure 7.7), decapsulation (Figure 7.8), fixed-weight sampling (Figure 7.9) and key generation (Figure 7.10). These diagrams also allow us to identify bottlenecks in the design, which can in turn be the target for future improvements.

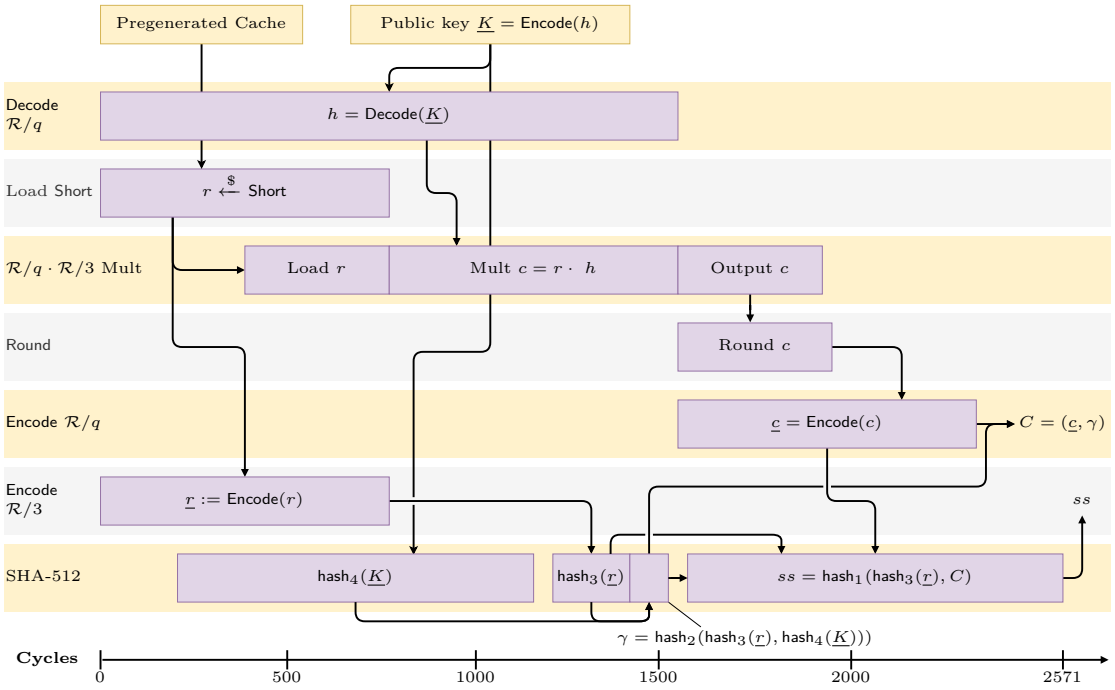


Figure 7.7.: Operation scheduling during Streamlined NTRU Prime encapsulation.

In Figure 7.7, a detail that is immediately visible is the high utilization of the SHA-512 module: It is active for almost the entire duration of the encapsulation. Even with perfectly optimal scheduling and an optimal cycle count for a SHA-512 round (i.e., 80 cycles per 1 024 bit input block), we can conjecture that the speed of the SHA-512 module will likely set the absolute lower limit of Streamlined NTRU Prime hardware implementations. For this, let us look at the encapsulation in Algorithm 2 for all the objects that must be hashed, their size in bytes for the sntrup761 parameter set, and minimum number of cycles the hashing could take:

- Hash of the encoded public key $\underline{K}$ → 1159 bytes → 800 cycles

- Hash of the encoded $\underline{r} \rightarrow 192$ bytes $\rightarrow 160$ cycles. This operation can in theory be done in parallel to the hashing of the public key, assuming a second SHA-512 module
- Confirmation hash $\gamma \rightarrow 65$ bytes $\rightarrow 80$ cycles
- Shared secret hash $\rightarrow 1217$ bytes $\rightarrow 800$ cycles

Thus, even with absolutely perfectly optimized scheduling, the hashing of all objects would take 1680 cycles. We thus conjecture that this cycle count is the theoretical lower bound for *any* hardware implementation of Streamlined NTRU Prime for the sntrup761 parameter set. Our achieved cycle count of 2628 is thus only 1.56 times the theoretical lower limit. Any further improvements beyond this limit would require exchanging the SHA-512 module with a different hash function. A good candidate for this would be a Keccak based hash function [195] due to its excellent performance in hardware. However, this would break interoperability with the Streamlined NTRU Prime specification, as well as slow down software implementations [191].

Beyond the SHA-512 module, the next target for additional optimization should be the $\mathcal{R}/q$ decoder. The decoding of the public key takes more than half of the entire duration of the encapsulation and accelerating it would allow the $\mathcal{R}/q$ multiplier to in turn begin its operation sooner. In addition, the overall structure of the decoder is well suited for vectorization, though doing so would likely increase the number of DSPs of the decoder. However, because the DSP in the decoder only implements constant multiplications, implementing the multiplications with LUT would be an efficient alternative.

In contrast to the encapsulation, we can see in Figure 7.8 that SHA-512 is not a bottleneck during decapsulation. In fact, the SHA-512 module has significant idle times, even with the pre-hashing of the rejection ss'. Instead, the module that has the highest activity is the $\mathcal{R}/q$ polynomial multiplier. The majority of its active time is actually spent loading and unloading data, and not on the actual multiplication. However, increasing the loading and unloading speed is not trivial, as the other modules would also require modifications. For the $\mathcal{R}/q$ multiplier concretely this would include the rounded $\mathcal{R}/q$ decoder, the $\mathcal{R}/3$ polynomial multiplier, and modulo 3/rounding module and the rounded $\mathcal{R}/q$ encoder. We can also see the benefit of a dedicated $\mathcal{R}/3$ multiplier: Without such a module, the $\mathcal{R}/3$ multiplication would also have to be performed by the $\mathcal{R}/q$ multiplier, further increasing the bottleneck and adding an additional delay. Instead, the $\mathcal{R}/3$ multiplication is overlapping with the output and input of the $\mathcal{R}/q$ multiplier, reducing idle time.

A further possible optimization is the use of *multi-step* LFSR for the $\mathcal{R}/3$ multiplier. Instead of reading one coefficient from the BRAM and shifting the LFSR by one per cycle, it is possible to read two or more coefficients, and shift the LFSR by the same number. This in turn requires two (or more) times the amount of MAC units. This

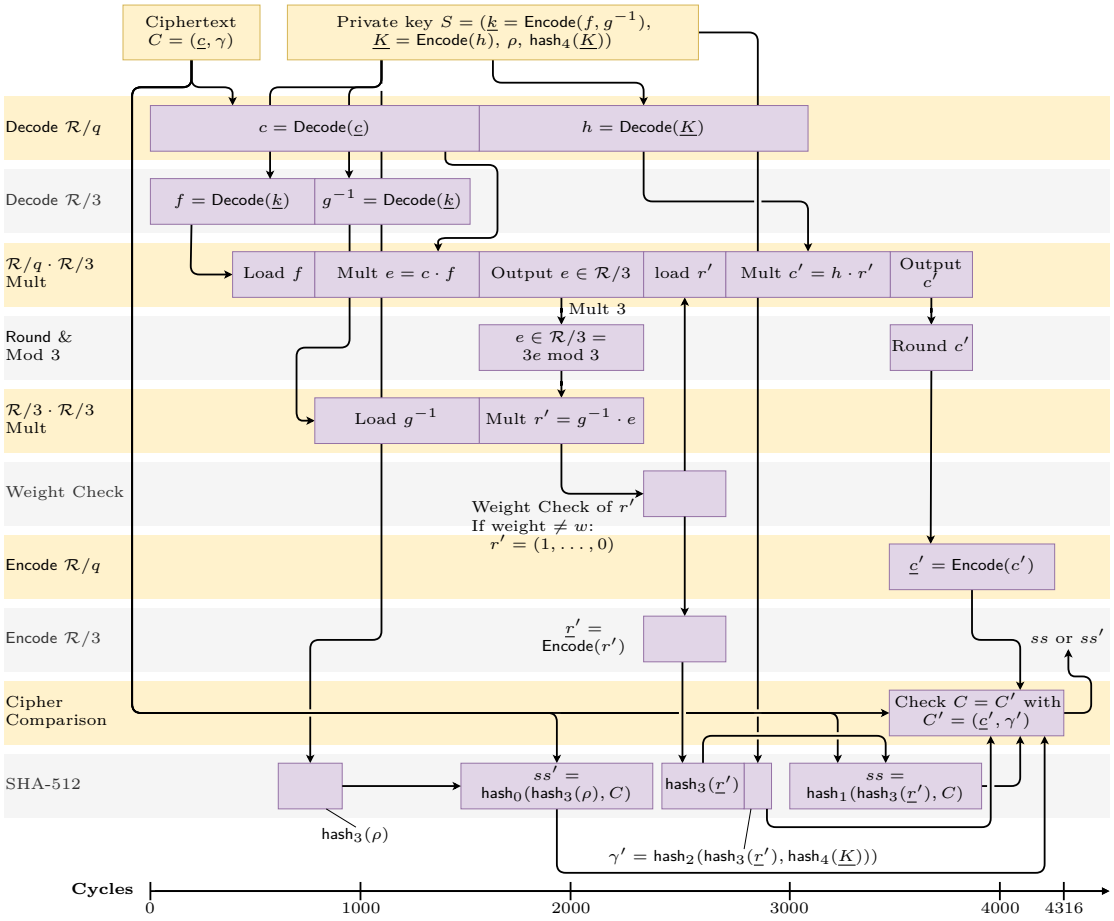


Figure 7.8.: Operation scheduling during Streamlined NTRU Prime decapsulation.

can be seen an unrolling of the outer loop in Line 2 in Algorithm 5, and was analyzed for the multiplier in Saber in [88]. There, the unrolling led to a speedup of the multiplication, but also to a significant area increase, as well as decreasing the maximum clock frequency. One could adapt the $\mathcal{R}/3$ multiplier to also support this type of loop unrolling. It would also be possible to apply the same technique for the $\mathcal{R}/q$ multiplier, however the area increase would be extreme, making it unlikely to be worth the speedup. Finally, like encapsulation, decapsulation would also benefit from a faster $\mathcal{R}/q$ decoder, as it would allow the multiplication of c and f to start sooner. However, a faster $\mathcal{R}/q$ decoder would only benefit the decoding of the ciphertext, and not the public-key for re-encryption because the public key decoding is done completely in the background during the decryption.

Figure 7.9 shows the fixed-weight sampling of short polynomials during multiple subsequent encapsulation runs for sntrup761. The pregeneration, sorting and subsequent

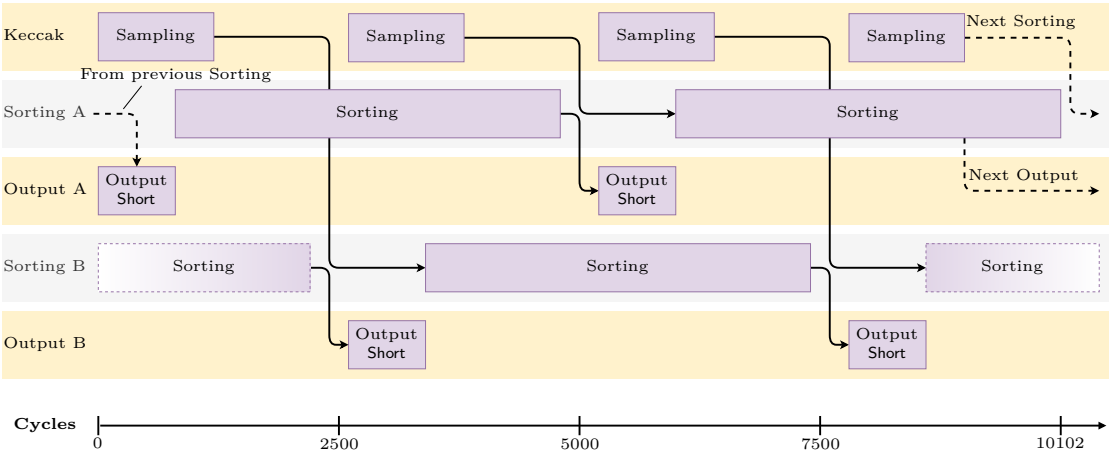


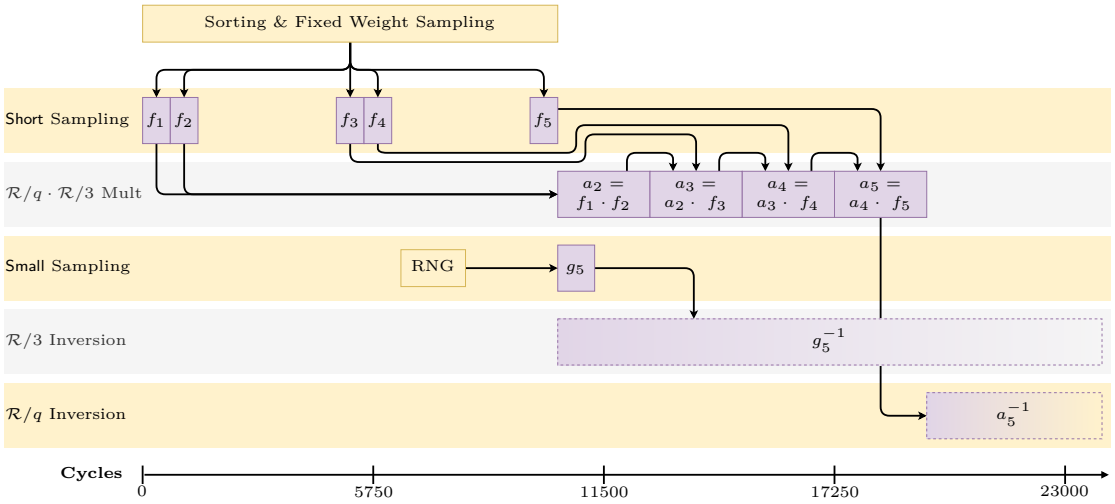
Figure 7.9.: Operation scheduling during Streamlined NTRU Prime fixed-weight sampling of short polynomials via sorting. Two sorting modules are present (labeled A and B), in order to generate short polynomials at a sufficient rate for the encapsulation. The boxes with a gradient continue beyond the area of the diagram.

caching of short polynomials is also visible, as well as the two sorting modules, labeled A and B. We can see the Keccak RNG is more than fast enough to generate the randomness for the sorting. The dual sorting modules can create two short polynomials over a period of 4 838 cycles. This averages to 2 419 cycles per short polynomial. This is sufficiently fast because an encapsulation takes 2 628 cycles.

In Figure 7.10, we depict the operation scheduling for the key generation for sntrup761, with batch inversion for a batch size of 5. We chose to show a batch size of 5 as it allows the figure to still be readable. We also split the figure into two parts, the preprocessing before the $\mathcal{R}/q$ inversion operation in Figure 7.10a, and the postprocessing after the inversion in Figure 7.10b. Including the $\mathcal{R}/q$ inversion operation itself would be of little benefit, as it would make the figure difficult to read. We also omit the hashing of the encoded public key as well as the sampling of the random ρ. We can see that the preprocessing takes little time overall. Roughly 50% of the preprocessing is spent on the sorting for the short sampling, the rest is spent in the multiplication module. In contrast, the postprocessing after the $\mathcal{R}/q$ inversion is complete takes significantly more time. The bottleneck of the postprocessing is the $\mathcal{R}/q \cdot \mathcal{R}/q$ polynomial multiplication. Other operations, such as the encoding of the public key or the $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplications are nearly negligible. While the $\mathcal{R}/q$ inversion on paper has the longest latency, the use of batch inversion allows us to amortize the cost of the single inversion over multiple keys. As a result, any further optimizations in future work should first target the $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT polynomial multiplier. The sampling and inversion of the g_i polynomials happens in parallel to the $\mathcal{R}/q$ inversion. The g_i inversion does not employ batch inversion because the inversion is already fast enough: All g_i inversions are complete before the

postprocessing steps begins. The $g_i{}^{-1}$ are buffered in memory until they are needed to compute h_i and be encoded into the private key.

(a) Key generation part before the $\mathcal{R}/q$ inversion.



(b) Key generation part after the $\mathcal{R}/q$ inversion.

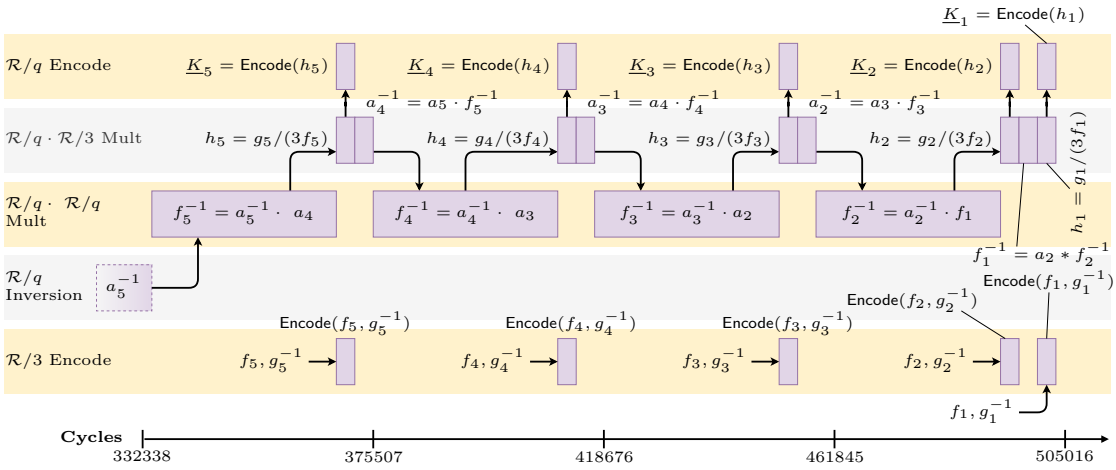


Figure 7.10.: Operation scheduling during Streamlined NTRU Prime batch key generation for a batch size of 5. The boxes with a gradient continue beyond the area of the diagram.

In addition, we can also see in Figure 7.10 how we interleave the final steps of the key generation with the batch inversion: Rather than completing the batch inversion and storing all $a_i^{-1}, f_i^{-1}\ \forall i$, we use each a_i^{-1} to compute f_i^{-1}, followed by the computation of the corresponding public key h_i. We then encode all relevant polynomials and output the public and private key, completing a key generation. Because we then no longer

need a_i^{-1} and f_i^{-1}, we can overwrite a_i^{-1} with a_{i-1}^{-1} and f_i^{-1} with f_{i-1}^{-1} in the next step of Montgomery's trick. We thus need less memory overall, reducing one of the major costs of performing batch inversion.

Part III.

Side-Channel Resistant Implementation

8. Gadget-Based Masking of Streamlined NTRU Prime

In this section, we will present the gadget-based masked implementation of Streamlined NTRU Prime, first published in [20]. The implementation is a joint work with the Ruhr University Bochum, with the initial concept being from Georg Land. We start with conceptual considerations of masking Streamlined NTRU Prime, followed by a concrete description of the gadget-based masked implementation. We will also describe our case study of applying the automated masking tool AGEMA to Streamlined NTRU Prime.

8.1. Conceptual Considerations of Gate-Level Masking

To implement the decapsulation as shown in Algorithm 3, we essentially need six major side-channel protected modules:

1. Polynomial multiplication with operands in $(\mathcal{R}/q, \mathcal{R}/3)$ and return values in $\mathcal{R}/q$,
2. Polynomial multiplication with operands in $(\mathcal{R}/3, \mathcal{R}/3)$ and result in $\mathcal{R}/3$,
3. Reduction component modulo 3,
4. Weight check component,
5. Rounding module, and
6. SHA-512.

Standard Approach Usually when we wish to mask polynomial multiplication modules we would apply additive masking. We would either instantiate multiple polynomial multipliers in parallel, or instantiate one polynomial multiplier that processes the shares consecutively. Moreover for Streamlined NTRU Prime, two of the three polynomial multiplications in the decapsulation have one public and one secret input, which can be realized very efficiently by applying additive masking because it only requires $d+1$ polynomial multiplications and no re-sharing. The $\mathcal{R}/3 \cdot \mathcal{R}/3$ multiplication, however, has two secret input polynomials. In order to perform a secure polynomial multiplication in the additive domain, $\frac{d^2+d}{2}$ fresh random polynomials need to be sampled. Additionally, $2(d^2+d)$ polynomial additions and d^2+d polynomial multiplications must be performed. In contrast, masking the reduction, weight check, and rounding is non-trivial in the arithmetic domain and would have to be solved in the Boolean domain. Finally, SHA-512 uses 64 bit additions, which is efficient in additive domain and feasible but less efficient in Boolean domain, as well as non-linear Boolean operations that strictly require Boolean masking.

In summary, this traditional approach is expected to yield a relatively efficient implementation at the cost of converting between additive and Boolean masking domain multiple times. Moreover, this type of implementation is often very specific in terms of masking degree, i.e., not parametrizable. Besides, the wide variety of applied techniques produces a larger attack surface, as shown in recent attacks on masking conversions [128, 129].

Applicability of Gadget-based Masking To overcome these downsides, we follow a recent line of research from the field of masking symmetric cryptographic schemes: *gadget-based* masking. For schemes in symmetric cryptography, we usually find a Boolean description that enables masking them at the gate level. This differs for public-key and post-quantum cryptography because these schemes typically employ arithmetic operations on number-theoretic structures such as multiplications in polynomial fields. Polynomial multiplications, however, consist of modular multiplications and additions in some finite number field. While the modular additions can be masked in Boolean domain through a secure adder, the modular multiplications are vastly more complex and are deemed infeasible to be masked in the Boolean domain due to the very high number of non-linear gates.

However, for Streamlined NTRU Prime, we have already observed in Section 6.2 that the three polynomial multiplications each have at least one factor in $\mathcal{R}/3$. Consequently, if we employ schoolbook multiplication, the underlying coefficient multiplication-accumulation has an input from $\mathbb{Z}_q$ being multiplied with either 1, 0, or -1 and then accumulated to another value in $\mathbb{Z}_q$. As we have also observed in Section 6.2, we do not have to perform a modular multiplication in this case. Instead, we can securely multiplex between the public input coefficient from $\mathbb{Z}_q$, its precomputed additive inverse, and zero. The result is added securely to the accumulation value using a masked adder. As indicated before, all other operations of Streamlined NTRU Prime are already feasible in the Boolean domain, enabling the first fully Boolean masked implementation of a public key and post-quantum secure scheme.

In the following, we describe our design considerations for each module in the Boolean domain. Note that in contrast to conventional hardware development, where it is desirable to have as many NAND gates as possible as they are the cheapest gate, the design goal in our case is to have as few as possible SecAND gadgets, as they require fresh randomness. Throughout our design, we use the HPC2 SecAND gadget [166], described in Section 5.2.3 and Algorithm 8.

8.1.1. Polynomial Multiplication

Polynomial multiplications are one of the most expensive operations in the decapsulation of lattice-based KEMs, and improving their performance has been a target in the literature and in this thesis [10, 17, 105, 106]. In addition to performance, we focus here

on achieving a *side-channel secure* implementation. During decapsulation, two types of multiplications are required:

- Multiplication in $\mathcal{R}/q$ with one operand from $\mathcal{R}/3$ (Lines 4 and 9 in Algorithm 3) and
- Multiplication in $\mathcal{R}/3$ (Line 5 in Algorithm 3).

8.1.1.1. Multiplication in $\mathcal{R}/q$

We observe that if we employ a standard schoolbook multiplication approach for both occasions of this multiplication, no coefficient multiplier is necessary. Instead, we use a secure adder and a secure three-way multiplexer. It is important to note that for both multiplications in $\mathcal{R}/q$, the input polynomial from $\mathcal{R}/q$ is public, while the other factor from $\mathcal{R}/3$ is secret. Thus, the idea is to compute the additive inverse of the input coefficient from $\mathcal{R}/q$, which is unmasked. Then, we securely multiplex with a masked select signal between both values and zero, and finally accumulate the result securely to the (intermediate) result coefficient. The architecture is shown in Figure 8.1.

Secure Multiplexing For this, we need a secure three-way multiplexer. The three *public* input signals are $z = 0, a_p = a, a_n = q - a \in \mathbb{Z}_q$. However, here we view them as Boolean values in $\mathbb{F}_2^{13}$. The secret select signal is $(f[1], f[0]) \in \{(0,0),(0,1),(1,1)\}$. We perform two consecutive secure 2-input multiplexing operations:

$$\begin{aligned} x[0]^{(0:d)} &= a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge \overline{f[1]^{(0:d)}} \\ &= a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge (f[1]^{(0:d)} \oplus 1) \\ &= a_p \wedge f[1]^{(0:d)} \oplus a_n \wedge f[1]^{(0:d)} \oplus a_n \\ &= ((a_p \oplus a_n) \wedge f[1]^{(0:d)}) \oplus a_n \qquad (8.1) \\ x[1]^{(0:d)} &= x[0]^{(0:d)} \wedge f[0] \oplus z \wedge \overline{f[0]^{(0:d)}} \\ &= ((x[0]^{(0:d)} \oplus z) \wedge f[0]^{(0:d)}) \oplus z \\ &= x[0]^{(0:d)} \wedge f[0]^{(0:d)} \qquad (8.2) \end{aligned}$$

Note that the public inputs can be set as the first share, while all other shares are just zeros. This is the reason why we can simply omit z in Equation 8.2. The SecAND gadget generates a uniformly random output also for the case that $(f_1, f_0) = (0,0)$.

Secure Addition Parallel prefix adders can achieve efficient addition in hardware. These concepts also have been adapted to the Boolean masked domain first in [212]. This was followed by a broader examination of more recent techniques like threshold implementation and gadget-based masking in [213], which we also deploy for our work.

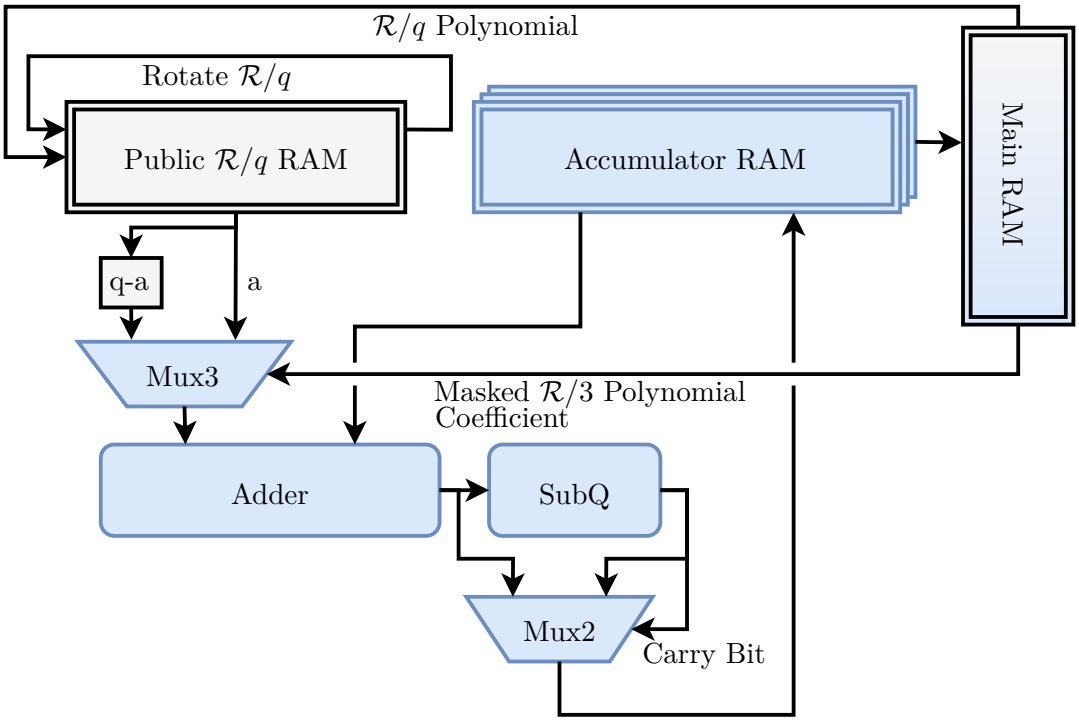


Figure 8.1.: Architecture of the masked $\mathcal{R}/q$ polynomial multiplier. Blue modules operate on masked shares. We have omitted the address calculation.

8.1.1.2. Multiplication in $\mathcal{R}/3$

For coefficient multiplications in $\mathcal{R}/3$, only 9 possible input combinations with three output combinations exist. Thus, we develop a direct Boolean masking utilizing the fact that the single inputs have a limited range. Multiplying two signed 2 bit coefficients $e[1:0] = e[0] - 2e[1]$ and $v[1:0] = v[0] - 2v[1]$ to a signed 2 bit value $r[1:0] = r[0] - 2r[1]$ can be done as follows:

$$r[0]^{(0:d)} = e[0]^{(0:d)} \wedge v[0]^{(0:d)} \tag{8.3}$$

$$r[1]^{(0:d)} = e[0]^{(0:d)} \wedge v[0]^{(0:d)} \wedge (e[1]^{(0:d)} \oplus v[1]^{(0:d)}) \tag{8.4}$$

Then, we add $r[1:0]^{(0:d)}$ to the accumulation value $a[1:0]^{(0:d)}$ and map the result back to the signed $a'[1:0]^{(0:d)} \in \{-1, 0, 1\}$ which can be done with the following formulas

that take into account that only $00_2, 01_2, 11_2$ are valid inputs:

$$\begin{aligned} a'[0]^{(0:d)} &= \left(r[0]^{(0:d)} \oplus a[0]^{(0:d)}\right) \\ &\quad \vee \left(r[0]^{(0:d)} \wedge \left(\overline{r[1]^{(0:d)} \oplus a[1]^{(0:d)}}\right)\right) \end{aligned} \tag{8.5}$$

$$\begin{aligned} a'[1]^{(0:d)} &= \left(r[1]^{(0:d)} \wedge \overline{a[0]^{(0:d)}}\right) \\ &\quad \oplus \left(\overline{r[1]^{(0:d)}} \wedge \left(r[0]^{(0:d)} \oplus a[1]^{(0:d)}\right)\right) \end{aligned} \tag{8.6}$$

8.1.1.3. Schoolbook Polynomial Multiplication

The schoolbook polynomial multiplication requires us to rotate one of the polynomials, in order to perform the polynomial reduction (see also Line 6 in Algorithm 5). Generally, there are three approaches for this: Either we rotate one of the input polynomials or the output polynomial. For our two "big×small" multiplications in $\mathcal{R}/q$, we have a small secret input represented by $2(d+1)$ bit, a big public input represented by $\lceil \log_2 q \rceil$ bit, and a big secret output represented by $(d+1)\lceil \log_2 q \rceil$ bit. Since shifting large amounts of data is expensive in terms of routing, FF demand, and dynamic power consumption, the natural choice is to rotate either of the input polynomials.

8.1.1.4. Polynomial Reduction Modulo $x^p - x - 1$

For the schoolbook multiplication, we can directly perform the polynomial reduction using the same LFSR rotation as with our unmasked multipliers in Section 6.2. We observe that $x^p \equiv x + 1 \bmod x^p - x - 1$, which indicates that the uppermost coefficient (x^p) during the rotation must be additionally added to the before lowermost coefficient. As we indicated before, we want to rotate either of the input polynomials. Applying this strategy to the $\mathcal{R}/3$ polynomial would increase the coefficient range to $[-2, 2]$ due to the extra addition during polynomial reduction in the same manner as for unmasked $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplier described in Section 6.2. We would then require a secure 5-way multiplexer instead of a secure 3-way multiplexer, increasing both area and randomness demand. In addition, the coefficient adder would have to be a secure 3 bit adder, which is more expensive than a unmasked $\mathbb{Z}/q$ adder. Thus, we choose to rotate the public $\mathcal{R}/q$ input polynomial and perform polynomial reduction in the same domain. This can also be seen in Figure 8.1.

8.1.2. Modular Reductions

For Streamlined NTRU Prime decapsulation, we require two different modular reductions.

8.1.2.1. Reduction Modulo q

This reduction is only applied for the accumulation within the $\mathcal{R}/q$ polynomial multiplications. We decide to use the non-negative modular representation in the interval $[0, q)$ only since we would need to check both for underflows and overflows in the centered

representation. Therefore, the value to reduce only grows by a maximum of one bit and can only provoke an overflow. Thus, a conditional subtraction by q suffices, resulting in two possible approaches:

1. We assume the accumulation produces a carry-out bit which we use to subtract q from the result conditionally. By this, our value always remains correctly in $\mathbb{F}_2^{13}$, but not necessarily in the interval $[0, q)$. Therefore, a final pass over the polynomial is required to reduce it to the minimal interval.
2. We subtract q from all accumulation results and obtain the carry bit from that operation. If this is 1, we know an underflow occurred. Thus, we can use the carry bit to securely multiplex between the original accumulation value and the subtracted value. This keeps all intermediate values in the minimal interval $[0, q)$. This can also be seen in Figure 8.1.

8.1.2.2. Reduction Modulo 3

For the modulo 3 reduction, we have an input from $\mathbb{Z}/q$ and want to reduce it to $\{-1, 0, 1\}$. We can use the same technique as described in Section 6.7.2.6, with the sole difference that all bit operations are the secure version operating on masked shares.

8.1.3. Weight Check

Let $r'[0:1, 0:p-1]^{(0:d)}$ be an array of p shared 2 bit numbers. Valid values are $(0,0)$, $(0,1)$, $(1,1)$ if the signed representation is used, and $(0,0)$, $(0,1)$, $(1,0)$ for the unsigned representation. We wish to check if exactly w array elements are non-zero. Thus, the basic idea depends on the chosen representation.

Signed. We accumulate all $r'[0,:]^{(0:d)}$ values together, with a secure $\lceil \log_2 w \rceil$ bit adder.

Unsigned. We compute $r'[0,:]^{(0:d)} \vee r'[1,:]^{(0:d)}$ and accumulate the resulting shared bit vector with a $\lceil \log_2 w \rceil$ bit adder.

It follows that the signed representation demands fewer non-linear Boolean operations. For the secure adder, the same adder as used for the polynomial multiplications is applied.

Following this, we then bit-wise XOR the shared adder output with the public target weight w, and then OR all bits of the result together to a single shared result bit. The overwriting of r' can be performed with a secure 2-way multiplexer deciding between the secret r' and the fixed public vector $(1, 1, \ldots, 1, 0, 0, \ldots, 0)$.

8.1.4. Rounding

For rounding, we first perform a reduction of the coefficient modulo 3 and then subtract the result from the original coefficient. As a result of the modulo operation, we obtain two masked bits $a[1:0]^{(0:d)} \in \{(0,0), (0,1), (1,1)\}$. With this, we want to

1. add 1 for $a[1:0]^{(0:d)} = (1,1)$
2. add $q-1$, which is analogue to subtracting 1, for $a[1:0]^{(0:d)} = (0,1)$, and
3. add zero for $a[1:0]^{(0:d)} = (0,0)$

One way to achieve this is by multiplexing securely between $q-1$, 1 and zero depending on $a[1:0]^{(0:d)}$, which in turn would include more non-linearity. To avoid this, we construct the value $(a[0]^{(0:d)} \oplus a[1]^{(0:d)}) \cdot (q-1) + a[1]^{(0:d)} \bmod q$. In other words, this value consists of $a[0]^{(0:d)} \oplus a[1]^{(0:d)}$ in all binary positions where $q-1$ is 1, with the least significant bit consisting of $a[1]^{(0:d)}$. All other positions are zero. Doing so requires no additional non-linear gates. This value is then securely added to the original coefficient to perform the rounding. For the addition, we can re-use the addition-reduction procedure as used for polynomial multiplication.

8.1.5. SHA-512

As explained in Section 4.6, SHA-512 implements a round function using an adder (modulo 2^{64}), the two functions Σ_0 and Σ_1 and the functions SHA-Ch and SHA-Ma. The full message schedule array is computed with an adder (modulo 2^{64}) and the two functions S and R. The four functions Σ_0, Σ_1, S and R consist of simple shift or rotate operations by different values for each function. The outputs of the shifts and rotates are added together by XOR operations. All four operations are thus fully linear. SHA-Ch and SHA-Ma are non-linear functions processing E, F, G and A, B, C, respectively.

For our masked hardware implementation, we can secure the adders by applying the concept of the masked adder introduced in Section 8.1.1. We use a 64 bit adder to realize the correct addition. Masking Σ_0, Σ_1, S and R can be accomplished in a straightforward way since the shift and rotate operations do not introduce additional implementation overhead in hardware and all XOR gates can simply be replaced by secure XOR gadgets. Finally, SHA-Ch and SHA-Ma are bit-wise operations that can be implemented in parallel to match the width of the adder to be used. Hence, we can modify the formulas for both to reduce the number of non-linear gates in order to minimize the amount of required randomness and the area overhead leading to

$$\begin{aligned} \text{SHA-Ch}(E,F,G) &= (E \wedge F) \oplus (\overline{E} \wedge G) \\ &= (E \wedge F) \oplus ((E \oplus 1) \wedge G) \\ &= (E \wedge (F \oplus G)) \oplus G \qquad (8.7) \\ \text{SHA-Ma}(A,B,C) &= (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C) \\ &= (A \wedge (B \oplus C)) \oplus (B \wedge C). \qquad (8.8) \end{aligned}$$

8.1.6. Encoding, Decoding & Comparison

Streamlined NTRU Prime defines multiple en- and decoding algorithms for transforming polynomials in $\mathcal{R}/3$ and $\mathcal{R}/q$ to and from byte arrays [9] as described in Section 4.5.

Decoding the ciphertext and public key can be done unmasked because they are both public. We use our $\mathcal{R}/q$ decoder described in [17] (see Section 6.8). For decoding the secret polynomials f and g^{-1}, we also use our $\mathcal{R}/3$ decoder from [17] and apply masking afterwards. However, we also need to securely encode r' into a byte array to compute the confirmation hash and session key. For this, we apply masking to our $\mathcal{R}/3$ encoder from [17]. This is straightforward as the encoder only consists of a shift register and a 2 bit adder.

In the original algorithm specification, the recomputed ciphertext polynomial c' is encoded (line 10 in Algorithm 3) before the ciphertext comparison (line 14), using an $\mathcal{R}/q$ encoder. However, the $\mathcal{R}/q$ encoder requires a 16 bit multiplication (see Section 6.8) which would be prohibitively expensive to implement securely in the Boolean domain. We instead compare the ciphertext polynomial coefficients directly, after which we compare the confirmation hashes. This allows us not to implement the masked $\mathcal{R}/q$ encoder. The masked ciphertext comparison is straightforward: We do a bit-wise secure XOR of the two ciphertext coefficients and then repeatedly securely OR the output together.

8.2. Implementation of Gadget-Based Masked Streamlined NTRU Prime

After introducing the theoretical background of masking all required operations, we now discuss the implementations of each building block. This implementation was first presented in [20] as part of this thesis, and the code of the implementation is available at [21]. Afterwards, we briefly discuss the generation of fresh randomness that is necessary to achieve a side-channel protected implementation.

8.2.1. Building Blocks

In order to implement the operations described in Section 8.1, we define the following masked modules and described them in detail in the following sections:

Add13 pipelined 13 bit Sklansky adder with a carry-out for usage in polynomial multiplication, weight check and rounding

CSubQ pipelined 13 bit Sklansky adder with one operand fixed to the two's complement of q, with a subsequent multiplexer, for usage in the modular reduction to $\mathbb{Z}_q$

Mod3 pipelined reduction from 13 bit modulo 3

Mul3 pipelined $\mathbb{Z}_3$ multiplier for usage in the $\mathcal{R}/3$ polynomial multiplication

Mux3 pipelined 3-way multiplexer with public input $\{a, 0, -a\}$ and a secret select signal

Mux2 pipelined 2-way multiplexer with secret input and a secret select signal

SHA-Ch 64 bit wide SHA-Ch step

SHA-Ma 64 bit wide SHA-Ma step

Add64 pipelined 64 bit Sklansky adder for the SHA-512

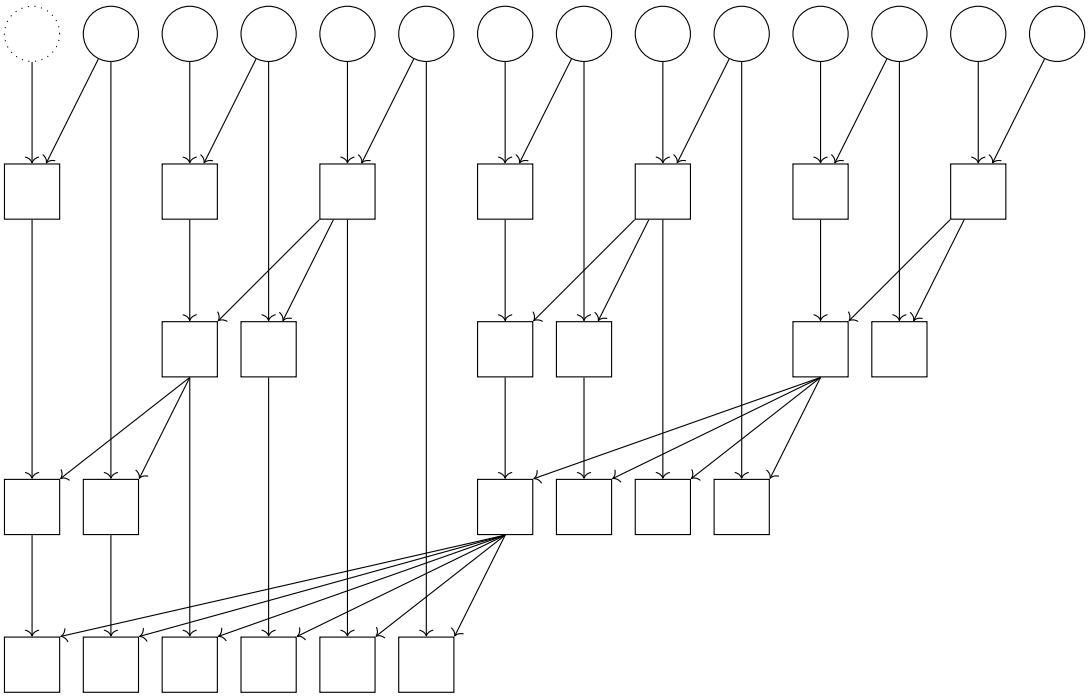


Figure 8.2.: 13 bit Sklansky adder construction with a carry-out.

8.2.1.1. Add13 and Add64

In [213], Bache and Güneysu compare the Brent-Kung, Kogge-Stone, and Sklansky adder architectures in the context of Boolean masking. For gadget-based masking, the Sklansky adder [214] turns out to be the optimal choice, having the same low latency as Kogge-Stone but less randomness demand while having a lower latency than Brent-Kung at the cost of slightly more randomness [213]. A higher latency increases the area cost for pipelined designs because we need more storage cells for the additional pipeline stages. In our case, all of our adders are fully pipelined.

The 12 bit Sklansky adder with carry-out deployed in our implementation is shown in Figure 8.2. We have a total of 5 layers. In the first layer, for input bits $a[i]^{(0:d)}, b[i]^{(0:d)}$

where $i \in \{0, \ldots, 12\}$, we compute in each circle:

$$g_1[i]^{(0:d)} = a[i]^{(0:d)} \wedge b[i]^{(0:d)} \tag{8.9}$$

$$p_1[i]^{(0:d)} = a[i]^{(0:d)} \oplus b[i]^{(0:d)} \tag{8.10}$$

Note that the dotted circle indicates the rightmost $p_1[13]^{(0:d)}, g_1[13]^{(0:d)}$, which are set to 0, as we only have a 13 bit input. It thus does not require any computation. It is needed to compute the carry-out, as we have a 14 bit output, and ensures that the lower layers operate on the correct inputs. Computations in the lower layers are represented by squares. Each square node of layer k has four inputs, the two "left" inputs $g_{kl}^{(0:d)}, p_{kl}^{(0:d)}$ and the two "right" inputs $g_{kr}^{(0:d)}, p_{kr}^{(0:d)}$, and computes the following outputs:

$$g_k^{(0:d)} = g_{kl}^{(0:d)} \oplus \left(p_{kl}^{(0:d)} \wedge g_{kr}^{(0:d)}\right) \tag{8.11}$$

$$p_k^{(0:d)} = p_{kl}^{(0:d)} \wedge p_{kr}^{(0:d)} \tag{8.12}$$

Finally, note that none of the leaf nodes in the lowest layer need to compute $p_5^{(0:d)}$, as only the final $g_5^{(0:d)}$ values are needed for the final output. The 14 bit output $O^{(0:d)}$ is computed as follows:

$$O[i]^{(0:d)} = g_5[i]^{(0:d)} \oplus p_1[i]^{(0:d)} \ \forall i \in \{1, \ldots, 13\} \tag{8.13}$$

$$O[0]^{(0:d)} = p_1[0]^{(0:d)} \tag{8.14}$$

The 64 bit adder works equivalently, though with a total of six levels. In this case, we do not need a carry-out because the additions in the SHA-512 module are all module 2^{64}.

8.2.1.2. CSubQ

For the conditional subtraction with q, we take a similar approach. We instantiate another Sklansky adder with one public operand fixed to the two's complement of q. Then, after each addition (let us denote the result here as $x^{(0:d)}$), we perform this subtraction by q and obtain $(q - x)^{(0:d)}$ as well as the shared carry-out bit. Using this bit, we multiplex securely between $x^{(0:d)}$ and $(q - x)^{(0:d)}$, selecting the former if the carry-out is one (indicating an underflow has occurred) and else the latter. The fixed input enables the synthesizer to optimizations significant parts of the adder circuit due to constant propagation.

8.2.1.3. Mod3

The architecture to compute this is depicted in Figure 8.3. For the secure additions and subtractions, we employ simple ripple-carry adders as parallel prefix adders have no advantage for these small bit widths. The Mod3 also has a special feature: its input is in the non-negative representation in the interval $[0, q)$, as this is what is output by the polynomial multiplier. At the same time, its output is in the signed representation in the interval $[-1, 1]$ because the weight check is more efficient for this representation (see Section 8.1.3).

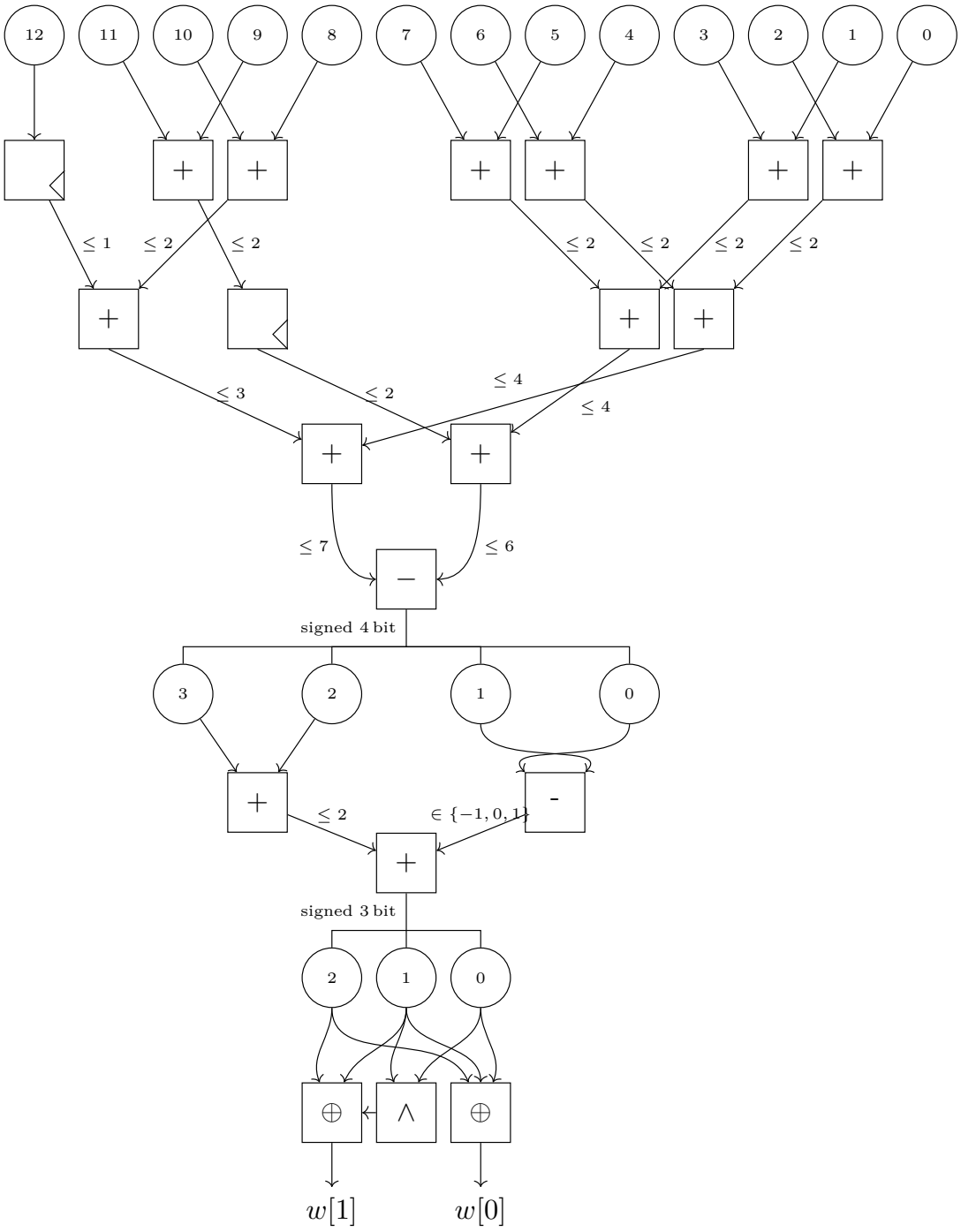


Figure 8.3.: Mod3 module [20].

8.2.1.4. Mux3 and Mux2

Mux3 can be implemented with three pipeline stages because the HPC2-SecAND gadget has a delay of two cycles for one input and one clock cycle for the other. We instantiate 13 of these 2 bit Mux3 in parallel in order to feed Add13 without idling.

We have a delay of two cycles for Mux2, which has two secret data inputs and a secret select input. We instantiate 13 Mux2 in the $\mathcal{R}/q$ multiplier to select between the CSubQ output and the non-subtracted value. We also instantiate two Mux2 during the weight check calculation to select between the original r' and the fixed vector. Finally, we use eight Mux2 to select between the encoded r' and ρ after the ciphertext comparison.

8.2.1.5. SHA-Ch and SHA-Ma

Both the SHA-Ch and SHA-Ma can be directly implemented according to Equation 8.7 and 8.8 respectively with the HPC2-SecAND gadget. We implement both operations with a width of 64 bit, in order to be able to directly feed the output to the Add64 module. The SHA-Ch has a latency of 2 clock cycles, while SHA-Ma has a latency of 3 clock cycles.

8.2.1.6. Mul3

The $\mathbb{Z}_3$ multiplier of the Mul3 module can also be directly implemented according to Equations 8.3 through 8.6 with the HPC2-SecAND gadget. The Mul3 module is fully pipelined, with a latency of 5 clock cycles.

8.2.2. Randomness Generation for Masking

The generation of a large amount of randomness needed by the secure gadgets is comparatively easy to solve on an FPGA or ASIC, but there are still different approaches available:

Pre-generated randomness. The first approach precomputes randomness and stores it in BRAMs. This approach does not require any LUT, though it requires a large amount of BRAM. In addition, it is only suitable for testing.

PRNGs using LFSRs. LFSRs can produce high quality statistical randomness, as long a suitable tap polynomial is chosen. This approach is very lightweight because LFSRs require very little area in hardware but does not create cryptographically secure randomness. However, for masking, statistical randomness may be sufficient, as shown in [215]. There, just three LUT were needed to create one random bit per cycle. At the same time, a recent work showed that using LFSRs to generate random bits for masking may be problematic in certain cases due to the linear nature of LFSRs [216].

True Random Number Generators (TRNGs). A TRNG creates randomness by sampling the noise of a physical process. An example of a TRNGs in an FPGA is the sampling of the jitter of a freely looping ring oscillator or Phase-Locked Loop (PLL) [217]. However, generating secure instances is non-trivial [218] and requires appropriator post-processing to remove bias. A TRNG also often provides only limited throughput.

Extendable Output Function (XOF)/ stream cipher. An XOF can provide large amounts of cryptographically secure randomness as long as it is properly seeded. A Keccak-based XOF is highly suitable for hardware implementations, as Keccak is very efficient in hardware (See also Sections 6.12 and 7.1.6). A similar approach is to use a stream cipher to generate random bits. In both cases, we can optimize the RNG by using round-reduced version of the algorithms. This approach was analyzed in the very recent work [216]. There, they found that a round-reduced variant of the stream cipher Trivium [219] – called Bivium B [220] – is optimal for generating large amounts of random bits for use in masking. Concretely, they report a cost of 22.9 GE/bit and 3.65 LUT/bit while generating 256 bit/cycle.

Due to the straightforwardness of randomness generation and in order to remain platform agnostic, we do not include a concrete RNG in our implementation.

8.3. Case Study: Applying AGEMA to Streamlined NTRU Prime

In this section, we will describe our case study of applying the AGEMA tool [22] to automatically create a masked implementation of Streamlined NTRU Prime from our low-area implementation [17]. This was done as part of the preparation prior to our gadget-based masked implementation in [20]. Ideally, this would allow us to create a secure masked implementation with minimal additional effort, as well as reduce the potential for implementation flaws that reduce security. Additional background on AGEMA is in Section 5.4.

Concretely, we apply AGEMA to the $\mathcal{R}/3 \cdot \mathcal{R}/3$ and $\mathcal{R}/q \cdot \mathcal{R}/3$ polynomial multiplier. For this, we synthesize our low-area $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplier from [17] and our $\mathcal{R}/3 \cdot \mathcal{R}/3$ multiplier from this work to an ASIC gate-level netlist, using the Nangate Opencell library [86]. We immediately encounter two problems:

1. AGEMA does not support control structures such as FSMs.
2. The Nangate library does not have any memory cells such as SRAM. As a result, any memories are left as black boxes, and are not supported by AGEMA.

To address point 1, we separate the control logic and data flow of both multipliers into separate files, and only apply AGEMA to the data flow logic. This includes the MAC unit (see Algorithm 10 and 9), the modular reduction modules (see Section 6.7.2.4)

and the LFSR tap points (see Figure 9). To address point 2, we manually modify the memories to the correct sizes to be able to store all shares. We apply AGEMA using the HPC2 gadget because this allows us to flexibly choose the masking degree. Another issue we encounter when applying AGEMA is that only the naive approach is reliable: For example, the transformation of the $\mathcal{R}/q$ MAC netlist into a BDD would not complete even after 848 hours of computation time, after which we decide to terminate the process. As a result, we decide to focus on the naive approach. We also use the pipelining feature of AGEMA, so that our masked modules can process new input every clock cycle in the same way as our unprotected modules.

Table 8.1.: Area results for the masked modules generated by AGEMA. All modules use HPC2 gadgets, the naive approach and with pipelining enabled.

Module	Area				Pipeline	max rand.	d
	LUT	FF	BRAM	DSP	Depth	bits / cycle	
$\mathcal{R}/q$ MAC	1 375	10 102	0	0	96	130	1
$\mathcal{R}/3$ MAC	190	368	0	0	16	18	1
$\mathcal{R}/q \cdot \mathcal{R}/3$ LFSR Tap	452	2 305	0	0	13	36	1
Mod $q = 4591$	8 296	24 642	0	0	78	825	1

We then synthesize the generated, masked modules for an FPGA, in order to gauge their area usage. The numbers for first-order security are listed in Table 8.1. We can immediately see that none of the modules are particularly efficient. The modular reduction stands out, requiring a significant number of both LUT and FF, as well as massive number of random bits per clock cycle. The cause for the high randomness consumption is the large amount of NAND gates that are used by the ASIC synthesis tool. This is normally a sensible strategy for ASIC synthesis, as NAND gates have the smallest area footprint of all gates. However, as explained in Section 8.1, the non-linear NAND gates are significantly more expensive then other gates such as XOR when it comes to masking them. In addition, the ASIC synthesis tool is unaware of the additional register stages caused by NAND gates, and instead chains them one after the other. This leads to the very high pipeline depth of the modules, which in turn leads to the high FF usage. Finally, the base design is also not amiable to masking: In our unprotected design, we rotate the secret $\mathcal{R}/3$ polynomial in the LFSR, used the signed representation for the accumulator register, and delay the modular reduction until the very end. While these design choices are beneficial for the unprotected design, they cause a significant overhead in the masked modules. Design choices that are more efficient for masking are described in Section 8.1.

As a result, we conclude that using AGEMA to mask Streamlined NTRU Prime is infeasible without significant modifications to the design architecture. This somewhat defeats the point of using an automated tool like AGEMA, as our starting goal was

to generate a masked implementation with minimal manual effort. In addition, an "AGEMA-aware" ASIC synthesis tool would be of interest for future work. Such a synthesis tool would take the overhead and register stages of masked NAND gates into account during the initial synthesis of the gate-level netlist. This would minimize both the number of non-linear gates as well as the total pipeline depth.

9. Evaluation & Discussion of the Gate-Level Masked Implementation

In this chapter, we present the implementation results of the gadget-based masked implementation of Streamlined NTRU Prime, first published in [20] as part of this thesis. Furthermore, we present a formal verification of our building blocks in order to demonstrate their protection against side-channel attacks. We also compare our hardware implementation of Streamlined NTRU Prime to a secure hardware design of Saber and Kyber. Afterwards, we discuss some of the potential improvements identified in our design. Finally, we analyze the application of our concepts and approaches to Kyber.

9.1. Implementation Results

We implement our design on a Xilinx Artix-7 device, using Vivado v2021.2 (64-bit), for the sntrup761 parameter set. We also synthesize our design for an ASIC using the 45nm Nangate open cell library [86]. Table 9.1 shows the latency, frequency, and peak randomness demand per module and masking degree. As can be seen there, the cycle count is dominated by the three polynomial multiplications, which take 93 % of all total cycles. At the same time, the peak randomness is always set by the 64 bit adder in the SHA-512 module. While the total cycle count is independent of the masking degree, the maximum clock frequency varies: On an FPGA and at masking degree 1 and 3, the design reaches 200 MHz, but the maximum frequency is lower for masking orders 2, 4, and 5. For all three, the critical path lies in the SHA-512 module. For the ASIC, the design reaches a higher maximum clock frequency than the FPGA at first order, with 207 MHz. However, as the masking degree increases, the maximum frequency drops off faster, reaching just 75 MHz at fifth order and 100 MHz at sixth order. Here, the critical path also lies in the SHA-512 module.

In Table 9.2, the area demand per module and masking degree is shown for Artix-7 FPGA. As expected, the area increases vastly with increasing masking degree. Interestingly, for all masking orders, SHA-512 dominates the resource cost consuming roughly 61 % of all LUT and FF. The next most expensive operation is the rounding during the re-encryption, followed by the $\mathcal{R}/q$ polynomial multiplication. When comparing the ratios of cycle counts and the resources consumed, it is apparent that the current SHA-512 implementation is sub-optimal: it is too expensive when considering the whole design. In particular, the full 64 bit adder is oversized. For a better ratio of cycles and resources consumed, using a smaller, 16 bit adder multiple times for each 64 bit addition, would be more efficient, while only adding a comparatively minor number of cycles. Doing so would also allow the SHA-Ch and SHA-Ma gadgets to have a smaller widths, saving

Table 9.1.: Latency, frequency, and randomness results after synthesis. Note that the cycle count for SHA-512 is for a single 1 024 bit block. We did not perform synthesis for orders sic and seven, as they no longer fit into an Artix-7 FPGA.

Module	Cycle Count	Maximum Randomness (bits per cycle)						
		Masking Degree						
		1	2	3	4	5	6	7
Decap	1 870 049	52	82	156	252	370	510	672
Encode $\mathcal{R}/3$	765	4	12	24	40	60	84	112
C' comp.	4 050	14	42	84	140	210	294	392
Decrypt	1 171 270	96	288	576	960	1 440	2 016	2 688
mod 3	29	46	138	276	460	690	966	1 288
Mul. $\mathcal{R}/3$	581 409	6	18	36	60	90	126	168
Weight calc.	9 145	42	126	252	420	630	882	1 176
Re-Encrypt	581 501	123	369	738	1 230	1 845	2 583	3 444
Rounding	812	123	369	738	1 230	1 845	2 583	3 444
Mul. $\mathcal{R}/q$	580 646	103	309	618	1 030	1 545	2 163	2 884
Adder 13 bit	10	32	96	192	320	480	672	896
13 Mux2	2	13	39	78	130	195	273	364
13 Mux3	3	26	78	156	260	390	546	728
SHA-512	7 845	310	930	1 860	3 100	4 650	6 510	8 680
SHA-Ma	2	128	384	768	1 280	1 920	2 688	3 584
SHA-Ch	2	64	192	384	640	960	1 344	1 792
Adder 64 bit	14	310	930	1 860	3 100	4 650	6 510	8 680
Total	1 870 049	310	930	1 860	3 100	4 650	6 510	8 680
FPGA	$f_{\max}$ **(MHz)**	200	182	200	169	179	–	–
	Latency (ms)	9.35	10.3	9.35	11.1	11.4	–	–
ASIC	$f_{\max}$ **(MHz)**	207	165	148	91	75	100	–
	Latency (ms)	9.03	11.3	12.6	20.5	24.9	18.7	–

further resources. Finally, this would reduce the maximum of random bits used per cycle.

When we compare the numbers of Table 9.2 and 9.1 with those from AGEMA in Table 8.1, we can clearly see the improvement of our hand-crafted design over the automatically generated design. The generated masked version of the modular reduction module alone uses more FF and randomness per cycle than our entire design.

In Table 9.3, we list the GE area demand per module and masking degree for an ASIC. As we did not have access to a memory macro, we list the memory footprint separately. We see similar behavior to the FPGA area demand, with the SHA-512 dominating the area demand, followed by the rounding during the re-encryption. The total GE also grows significantly as the masking degree increases, while the SRAM usage grows more slowly.

Table 9.2.: FPGA area results for the Xilinx Artix-7. Note that this does not include the area needed for randomness generation. Not listed is the DSP usage: 4 DSPs are needed as multipliers in the decoder, regardless of the masking degree.

Module	Masking Degree											
	1			2			3			4		
	LUT	FF	BR	LUT	FF	BR	LUT	FF	BR	LUT	FF	BR
Decap	2270	1180	4.5	2493	1575	6	3088	2256	6	3766	2980	8
Encode $\mathcal{R}/3$	61	52	0	77	80	0	104	115	0	131	157	0
C' comp.	278	263	0	503	530	0	855	895	0	1273	1358	0
Decrypt	1743	1602	0	2680	3225	1.5	4847	5451	1.5	7276	8282	1.5
mod 3	542	719	0	1197	1528	0	2274	2638	0	3474	4049	0
Mul. $\mathcal{R}/3$	470	208	0	329	319	1	489	476	1	665	675	1
Weight calc.	528	612	0	1066	1286	0	1947	2194	0	2941	3350	0
Re-Encrypt	2017	2450	0.5	4138	5180	1	7755	8936	1	11696	13714	1
Rounding	1888	2387	0	4080	5108	0	7695	8851	0	11636	13616	0
Mul. $\mathcal{R}/q$	1846	2148	1.5	3693	4419	2	6686	7554	2	9885	11553	2.5
Adder 13 bit	627	715	0	1352	1545	0	2523	2690	0	3729	4150	0
13 Mux2	182	221	0	390	468	0	676	806	0	1040	1235	0
13 Mux3	211	286	0	463	625	0	848	1107	0	1314	1723	0
SHA-512	11684	12035	2	22493	23880	3	38370	39406	8	56207	59097	9
SHA-Ma	1528	1664	0	3439	3840	0	6624	6912	0	10197	10880	0
SHA-Ch	896	1088	0	1920	2304	0	3584	3968	0	5440	6080	0
Adder 64 bit	5996	5663	0	12352	23162	0	22506	21770	0	32702	33740	0
Total	19923	19725	8.5	36340	39209	13.5	62498	65463	18.5	91731	98726	22
Total w/o SHA	8239	7690	6.5	13847	15329	10.5	24128	26057	10.5	35524	39629	13
SHA Pct.	58.4	61.0	23.5	61.9	60.9	22.2	61.4	60.2	43.2	61.3	59.9	40.9

Different Masking Degrees for Decrypt and Re-Encrypt In [172], the authors reason that re-encryption must be protected at a higher level than decryption during decapsulation (see also Section 5.3). Our design, and all building blocks can be easily adapted to any masking degree allowing a flexible configuration. However, doing so would decrease the modules that can be re-used across the design, such as the $\mathcal{R}/q$ multiplier which is used both during decryption and re-encryption.

9.2. Side-Channel Evaluation

In this section, we will evaluate the side-channel security of our masked Streamlined NTRU Prime implementation. The analysis was performed primarily by my co-authors from Ruhr University Bochum in [20]. We include it for reference and completeness. For better readability and consistency, we will continue to use "we" in this section.

In order to evaluate the protection against side-channel attacks, we rely on formal verification of each of our submodules, but additionally perform practical side-channel measurements based on Welch's t-test. Evaluating the entire decapsulation by practical measurements is currently infeasible for our side-channel setup due to the large amount of required clock cycles. In particular, the total number of samples required to

Table 9.3.: ASIC area results in Gate Equivalent (GE), using the 45nm Nangate open cell library [86]. The area does not include SRAM cells, which are listed separately. Note that this does not include the area needed for randomness generation. The area for the Encode $\mathcal{R}/3$ entity is not available for masking degrees one through three because it was merged with its parent entity.

Module	Masking Degree					
	1	2	3	4	5	6
Decap	14 703	18 520	23 632	29 561	37 176	46 078
Encode $\mathcal{R}/3$	n/a	n/a	n/a	1 130	1 447	1 799
C' comp.	2 052	4 103	6 943	10 571	14 981	20 208
Decrypt	14 727	28 021	46 047	68 449	95 889	128 306
mod 3	5 744	12 216	21 101	32 386	46 085	62 295
Mul. $\mathcal{R}/3$	2 452	3 436	4 756	6 065	8 025	10 412
Weight calc.	5 688	11 202	18 584	27 825	38 949	51 986
Re-Encrypt	29 615	56 009	90 348	127 595	176 907	234 225
Rounding	29 057	55 375	89 636	126 818	176 050	233 295
Mul. $\mathcal{R}/q$	25 244	45 906	73 131	103 820	143 784	190 601
Adder 13 bit	7 115	14 482	24 375	36 840	51 817	69 367
13 Mux2	1 607	3 510	6 144	9 511	13 611	18 442
13 Mux3	2 015	4 468	7 910	12 356	17 811	24 366
SHA-512	114 570	218 453	354 242	519 545	719 019	950 021
SHA-Ma	12 416	29 440	53 674	85 120	123 776	169 642
SHA-Ch	7 914	17 280	30 250	46 826	67 008	90 794
Adder 64 bit	55 503	114 820	195 205	296 601	419 131	563 160
Total	201 112	373 349	600 100	870 124	1 204 839	1 594 022
Total w/o SHA	86 542	154 896	245 858	350 579	485 820	644 001
SHA Pct.	57.0	58.5	59.0	59.7	59.7	59.6
SRAM (bits)	189 440	246 272	294 912	343 296	393 216	443 392

capture an entire decapsulation trace exceeds the memory buffer of our measurement setup. To this end, we formally verify the security of each module by using the verification tool VERICA [183], described in Section 5.5. We analyze our modules in the glitch-extended d-probing model for different security orders. The corresponding results are shown in Table 9.4. Note that all modules pass first- and second-order verification, while third-order verification is too complex for Mod3 and Mul3. For the Add13 and Add64 modules, we use the implementation by Bache and Güneysu [213], which is verified to be secure practically. We also do not verify the four functions Σ_0, Σ_1, S and R, as they consist purely of linear XOR operations which are trivial to implement.

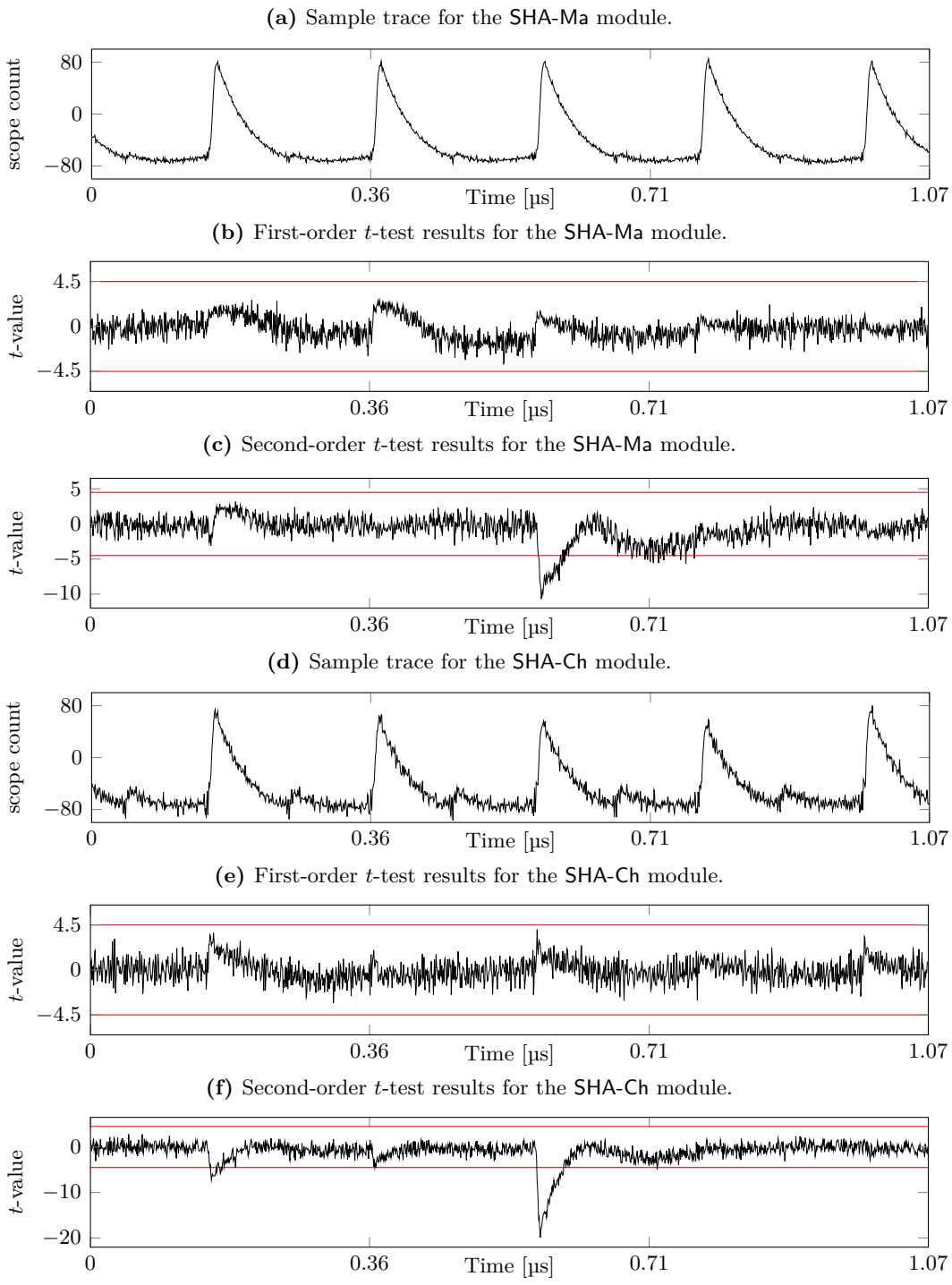
As an additional security analysis, we perform side-channel measurements of our first-order protected designs on a Sakura-G FPGA evaluation board, which is equipped with

Table 9.4.: Verification results of the protected submodules using VERICA. We report for each design the number of combinational gates, memory gates and the verification time. The verification of the expected security order is indicated by green check marks.

Module	First Order				Second Order				Third Order			
	comb.	mem.	sec.	time	comb.	mem.	sec.	time	comb.	mem.	sec.	time
Mux2	16	17	1✓	0.383 s	39	36	2✓	0.402 s	72	62	3✓	20.609 s
Mux3	28	31	1✓	0.385 s	72	69	2✓	0.521 s	136	122	3✓	4.985 h
Mod3	586	774	1✓	1.125 s	1464	1581	2✓	90.82 min	2742	2668	–	∞
Mul3	89	94	1✓	0.412 s	221	204	2✓	23.591 s	413	356	–	∞
SHA-Ch	16	17	1✓	0.404 s	39	36	2✓	0.420 s	72	62	3✓	26.355 s
SHA-Ma	28	26	1✓	0.386 s	72	60	2✓	0.928 s	136	108	3✓	11.5 h

a Xilinx Spartan 6 FPGA. The target FPGA is supplied with a 4 MHz clock while the power consumption is measured via the voltage drop over a 1 Ω shunt resistor. The power traces are acquired by using a ZFL-2000GH+ Low Noise Amplifier (LNA) connected to a Spectrum M4 oscilloscope (8 bit resolution). The oscilloscope collects the data with a sample rate of 1.5 GS/s. To generate the required online randomness, we instantiate a Keccak core used as Pseudorandom Number Generator (PRNG).

The measurement results for 10 million power traces can be found in Figure 9.1, Figure 9.2, and Figure 9.3. For all experiments, we first plot a sample trace to document a proper setup of the measurement equipment. In the subsequent plots, we use Welsh's t-test to detect potential leakage, as described in Section 5.5. We use the threshold of ± 4.5 to decide whether the design leaks information via the power consumption [180]. To this end, we do not observe any notable leakage in the first order in any of our modules but – as expected – some leakage in the second order.



(a) Sample trace for the SHA-Ma module.

(b) First-order t-test results for the SHA-Ma module.

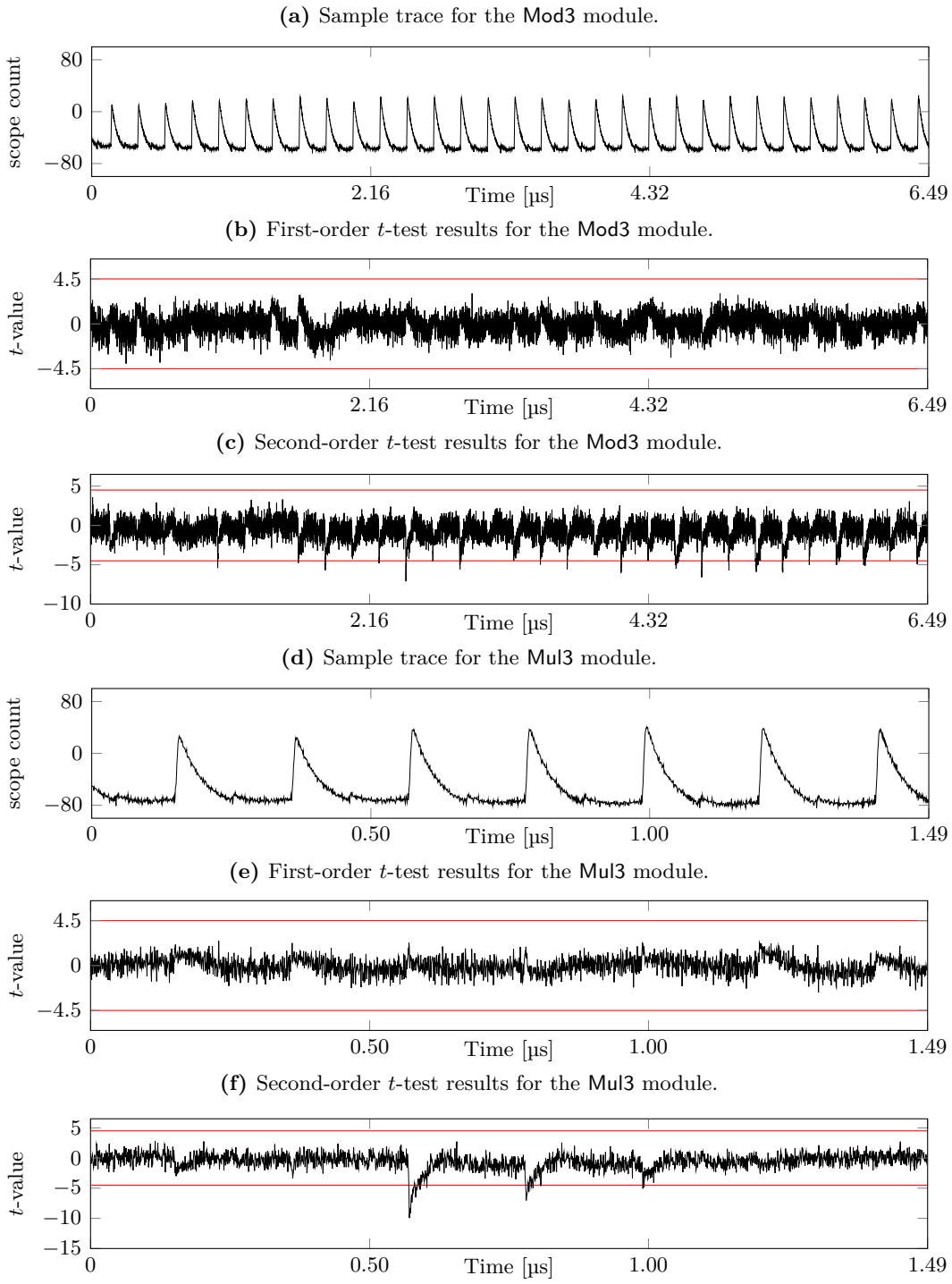
(c) Second-order t-test results for the SHA-Ma module.

(d) Sample trace for the SHA-Ch module.

(e) First-order t-test results for the SHA-Ch module.

(f) Second-order t-test results for the SHA-Ch module.

Figure 9.1.: Measurement results for the SHA-Ma module (**a**,**b** and **c**) and the SHA-Ch module (**d**,**e** and **f**) using 10 million traces. Both modules are instantiated for $d = 1$.



(a) Sample trace for the Mod3 module.

(b) First-order t-test results for the Mod3 module.

(c) Second-order t-test results for the Mod3 module.

(d) Sample trace for the Mul3 module.

(e) First-order t-test results for the Mul3 module.

(f) Second-order t-test results for the Mul3 module.

Figure 9.2.: Measurement results for the Mod3 module (**a**,**b** and **c**) and the Mul3 module (**d**,**e** and **f**) using 10 million traces. Both modules are instantiated for $d = 1$.

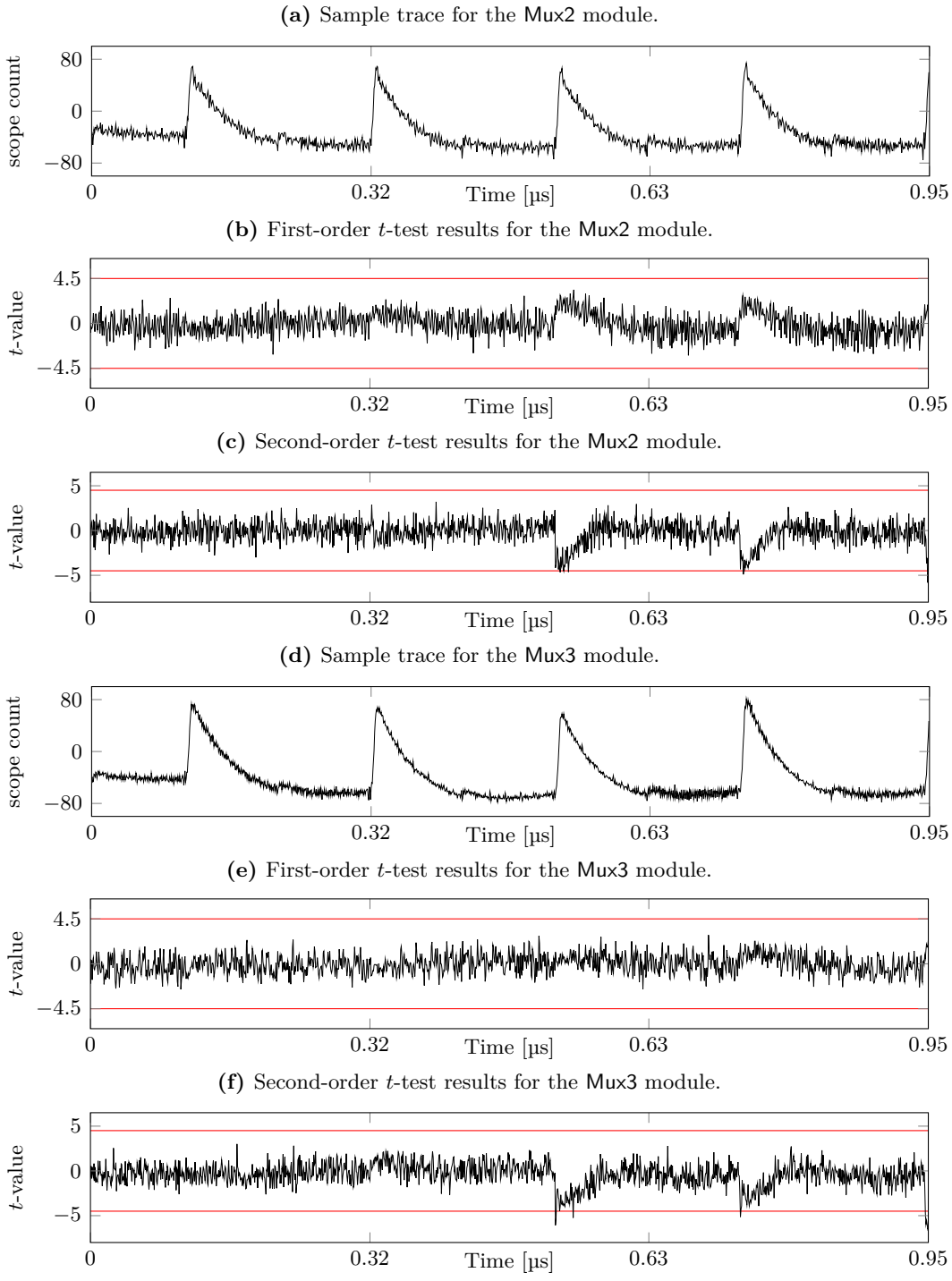


Figure 9.3.: Measurement results for the Mux2 module (**a**,**b** and **c**) and the Mux3 module (**d**,**e** and **f**) using 10 million traces. Both modules are instantiated for $d = 1$.

9.3. Comparison

In Table 9.5, we compare our implementation against our unmasked implementation of Streamlined NTRU Prime and two first-order masked FPGA implementations of Kyber [159] and Saber [160]. To the best of our knowledge, our implementation in [20] is the first higher-order full FPGA implementation of any PQC scheme and the first masked full ASIC PQC implementation. Previous ASIC designs were only hardware-software co-designs, such as Kyber and Saber in [158]. Although comparing full hardware designs with HW-SW co-designs is somewhat of an apples-to-oranges comparison, we still include the FPGA designs from [158] in Table 9.5 and compare their ASIC designs to ours in Table 9.6.

Table 9.5.: Comparison of our masked implementation both with unmasked Streamlined NTRU Prime, as well as masked Kyber and Saber [20]. All implementations are synthesized for the Artix-7 FPGA, except for [159], which is synthesized for the Virtex-7. The designs from [158] are RISC-V HW-SW co-designs, list the area including the RISC-V core, and list the total cycles count excluding the cycles to generate randomness.

Design	Area				Cycle count	f_{max} Mhz	max rand. bits / cycle	d	Ref.
	LUT	FF	BRAM	DSP					
sntrup761	36 789	22 700	3.5	9	10 989	137	0	0	**this**, [17]
sntrup761	6 279	3 086	3.0	7	85 628	131	0	0	**this**, [17]
sntrup761	43 446	26 123	2.5	5	4 316	187	0	0	**this**
sntrup761	19 923	19 725	8.5	4	1 870 049	200	310	1	**this**, [20]
sntrup761	36 340	39 209	13.5	4	1 870 049	182	930	2	**this**, [20]
sntrup761	62 498	65 463	18.5	4	1 870 049	200	1 860	3	**this**, [20]
sntrup761	91 731	98 726	22.0	4	1 870 049	169	3 100	4	**this**, [20]
Saber	19 299	21 977	0.0	64	72 005	125	DNR	1	[160]
Kyber-512	152 860	DNR	489.5	76	137 738	100	DNR	1	[159]
Kyber-512	29 889	17 152	52.5	13	929 072	58	DNR	1	[158]
Saber	29 889	17 152	52.5	13	905 395	58	DNR	1	[158]

As expected, our unprotected implementations (two “high speed” and one “low area”) are both smaller and faster. The masked Saber implementation has a comparable LUT and FF footprint to our first-order implementation and uses no BRAM but significantly more DSP. However, it is about an order of magnitude faster. In contrast, the masked Kyber-512 implementation is vastly bigger even than our fourth-order implementation, but only faster by a factor of 6.8 compared to our first-order implementation. Moreover, both the Saber and the Kyber-512 implementations only support first order, while our design can easily be instantiated at an arbitrary level, allowing protection against more advanced attacks. Our masked design also runs at a higher clock frequency than both the masked Kyber and Saber implementation, with a similar clock frequency as our improved

Table 9.6.: Comparison of our masked ASIC implementation with a masked HW-SW co-design of Kyber and Saber

Design	Area		Cycle	f_{max}	max rand.	d	Ref.
	GE	Memory	count	Mhz	bits / cycle		
sntrup761	201k	189k bits	1 870 049	207	310	1	**this**, [20]
sntrup761	373k	246k bits	1 870 049	165	930	2	**this**, [20]
sntrup761	600k	294k bits	1 870 049	148	1 860	3	**this**, [20]
sntrup761	870k	343k bits	1 870 049	91	3 100	4	**this**, [20]
Saber	259k	922k GE	905 395	78.33	DNR	1	[158]
Kyber-512	259k	922k GE	929 072	78.33	DNR	1	[158]

high-speed design. Finally, our masked gadgets have been verified to be secure, and we do not need any masking conversion which may be used in future attacks.

When we compare our FPGA design to the HW-SW co-design of Kyber and Saber from [158] in Table 9.5, we can see that our design has roughly twice the cycle count. However, our much higher frequency leads to our design having a lower latency overall. In addition, our first-order design uses fewer LUT, FF and DSP and significantly less BRAM. For ASICs in Table 9.6, we have a similar picture: Our first-order design has a higher cycle count, but also a higher frequency, leading to an overall lower latency. Our design also has a lower GE. Unfortunately, we cannot compare the memory requirements, as the authors from [158] only reports the memory usage in GE. Because we do not have a memory cell for our ASIC synthesis, we can only report the memory usage in bits. A final important detail in this comparison we must remember is that the design from [158] implements a full RISC-V System on Chip (SoC), which invariably causes an overhead compared to our dedicated implementation. In addition, the flexibility of the HW-SW co-design with a full SoC allows them to implement multiple schemes and parameters sets in the same hardware. However, their design currently only supports first-order masking.

Table 9.7 shows how the ASIC performance progresses for our implementation and provides a comparison to gadget-based implementations of the Advanced Encryption Standard (AES) [22]. While the randomness overhead is independent of the scheme (it only depends on the gadget, which is HPC2 for all schemes in this table), differences can be observed for area and delay. Still, the relative area overhead is similar for Streamlined NTRU Prime and AES when we increase the masking degree, with our overhead being slightly higher than the byte-serial version and significantly lower than the round version of AES. However, the delay overhead is slightly worse for $d = 3$. The reason for this is likely the larger absolute area of Streamlined NTRU Prime, which causes additional routing delays due to the longer transmission lines as well as routing congestion. In addition, we would like to highlight that our design has a lower peak randomness demand

Table 9.7.: Comparison of our masked Streamlined NTRU Prime with gadget-based masked implementations of symmetric schemes on ASICs [20]. The overhead is given as the fraction between the current row and the previous row minus one.

		Utilization				**Timing**		
		Area	overhead	Rand.	overhead	Latency	overhead	
Scheme	d	[GE]	[%]	[bit]	[%]	[ns]	[%]	**Ref.**
sntrup761	1	201 112	—	310	—	9.05×10^6	—	**this**, [20]
	2	373 349	85.6	930	200	11.3×10^6	25.9	
	3	600 100	60.7	1 860	100	12.6×10^6	11.5	
AES (byte-serial)	0	3 263	—	0	—	189.2	—	[22]
	1	10 090	209	34	∞	4 311	2 188	
	2	17 649	74.9	102	200	5 434	26.1	
	3	27 026	53.1	204	100	5 537	1.88	
AES (round.)	0	9 906	—	0	—	20.35	—	[22]
	1	52 597	431	680	∞	201.9	892	
	2	131 631	150	2 040	200	236.6	17.2	
	3	246 924	87.6	4 080	100	250.5	5.86	

then the round version of AES, despite our design being a much larger and more complex PKE scheme.

9.4. Discussion

This section addresses and discusses potential improvements and the huge overhead introduced by masking the symmetric core in Streamlined NTRU Prime. Additionally, we briefly discuss applying our concepts and approaches to Kyber.

9.4.1. Gadget-Based Masking

There are several advantages in a gadget-based masked implementation. First, it is effortless to adapt to an arbitrary masking degree. This obviously reduces the time required for the development. Moreover, no masking conversion can be attacked since there is none. The masking conversion was the target in the attacks against a first-order and third-order masked Saber implementation [128, 129]. Additionally, exchanging the underlying gadgets with others with the same latency properties is usually straightforward. For example, it could be possible to achieve a fault-secure implementation easily by deploying the work from [221].

9.4.2. Potential Improvements

We leave several potential improvements as future work and address them here. The most expensive operation from the latency viewpoint is polynomial multiplication. The

two $\mathcal{R}/q$ multiplications take 62 % of the decapsulation cycle counts, and the multiplication in $\mathcal{R}/3$ takes another 31 %. To speed this up, it would be possible to instantiate more adders in parallel at the cost of slightly more area and a potentially higher peak amount of randomness per clock cycle, depending on the grade of deployed parallelism. Thus, halving the latency of both multipliers results in a 47 % speed-up at the cost of approximately 8 % more gate equivalents for the first-order ASIC implementation.

9.4.3. Improvements to CSubQ

The CSubQ module also has room for optimizations: While the fixed input already enables optimizations by the synthesizer, further improvements could be made by optimizing the adder architecture itself for a fixed operand. Since we know the positions of the zeros, we could simplify our adder as depicted in Figure 9.4. The computation of all p values below the first row is the same as before. However, we can completely omit computing the first row of p, g as described in Equation 8.9 and Equation 8.10. Instead, we know, given an input $a[12:0]^{(0:d)}$, for each circle in Figure 9.4 that

$$g[i]^{(0:d)} = \begin{cases} a[i]^{(0:d)} & \text{if } (-q)[i] = 1 \\ 0 & \text{else} \end{cases} \tag{9.1}$$

$$p[i]^{(0:d)} = \begin{cases} \overline{a[i]^{(0:d)}} & \text{if } (-q)[i] = 1 \\ a[i]^{(0:d)} & \text{else} \end{cases} \tag{9.2}$$

Like with the Add13 module, the dotted circle indicates the rightmost $p_1[13]^{(0:d)} = 1$ and $g_1[13]^{(0:d)} = 0$, as we only have 13 bit input. In Figure 9.4, the circles filled with the diagonal line pattern indicate that the fixed input bit of the two's complement of q is one. For the squares, Equation 9.1 allows us to further optimize Equation 8.11. We have four different cases now:

Non-filled Computed as before: $g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right)$

Grid $g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge 0\right) = g_l^{(0:d)}$

Dotted $g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = 0 \oplus \left(p_l^{(0:d)} \wedge 0\right) = 0$

Horizontal lines $g^{(0:d)} = g_l^{(0:d)} \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = 0 \oplus \left(p_l^{(0:d)} \wedge g_r^{(0:d)}\right) = p_l^{(0:d)} \wedge g_r^{(0:d)}$

The computation of $p_k^{(0:d)}$ (from Equation 8.12) are unchanged from the normal Sklansky adder. These optimizations would lead to an area reduction of the CSubQ module, as well as reducing the randomness demand due to the reduction of SecAND gadgets.

9.4.4. Improvements to the Symmetric Core

As discussed in Section 9.1, masking the symmetric core (i.e., SHA-512) in Streamlined NTRU Prime consumes a considerably large part of the entire implementation's footprint

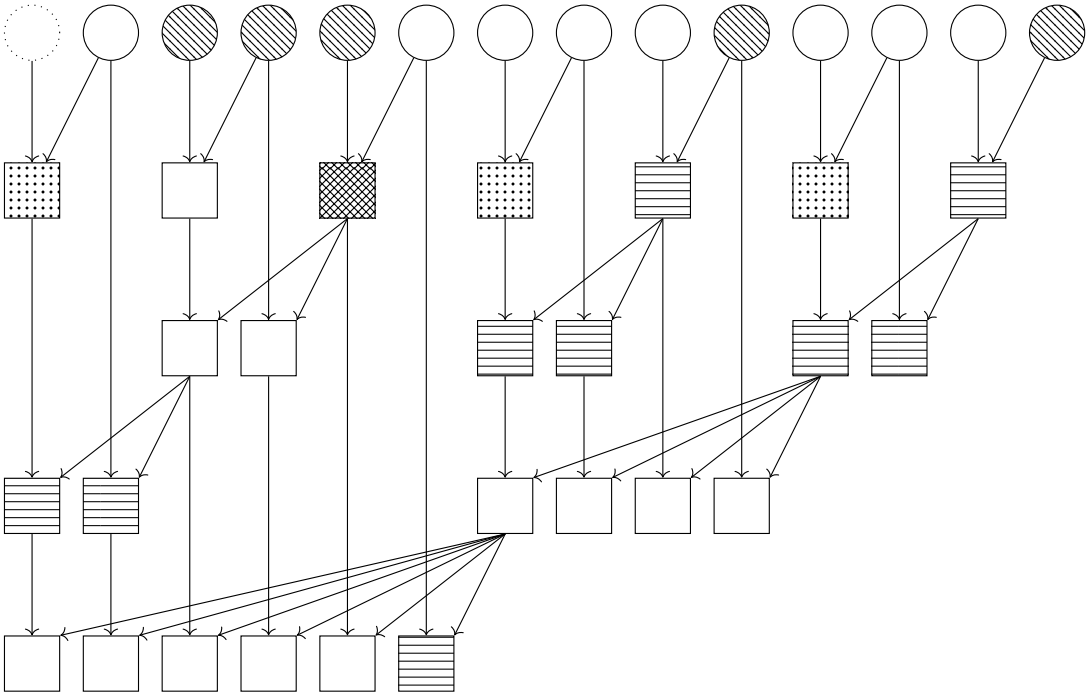


Figure 9.4.: CSubQ with optimizations for $q = 4591$ [20]

and has the highest per cycle randomness consumption. Nevertheless, secure and hardened SHA-512 implementations are widely deployed in industry and can, for example, be found in smartcards and secure elements [222]. Thus, one could assume that a secure SHA-512 is already available and does not need to be implemented. If we exclude the SHA-512 from the area consumption (see Table 9.2 and 9.3), then the design is not only surprisingly small at first order, but the area overhead is much more moderate with increasing masking degree.

Another possibility would be to replace the 64 bit Sklansky adder deployed in the SHA-512 module by a smaller one, trading area for latency, as already mentioned in Section 9.1. Moreover, it is possible to deploy no additional adder for the SHA-512 module by reusing the secure adder from the polynomial multiplication module. In this case, five consecutive 13 bit additions would yield the 64 bit addition. This would require cleverly scheduling the additions required by SHA-512 such that the 13 bit adder pipeline is maximally occupied. As can be seen from Table 9.3, the 64 bit Sklansky adder occupies about half of the area of the SHA-512 module and about a quarter of the overall area.

Additionally, in order to reduce the total area overhead introduced by the masked symmetric core in Streamlined NTRU Prime, the SHA-512 could be replaced by an implementation based on Keccak [195]. Because Keccak does not use an adder internally,

it is significantly easier and cheaper to mask. Most notably, it can be implemented with a very low amount of fresh randomness [223]. In addition, as the critical path lies in the SHA-512 module for both FPGAs and ASICs, using Keccak would likely increase the maximum achievable clock frequency. However, this would deviate from the Streamlined NTRU Prime specification and would not be interoperable with other Streamlined NTRU Prime implementations.

9.4.5. Applicability to Encapsulation and Key Generation of Streamlined NTRU Prime

Many of our secure modules can be re-used to implement the encapsulation and key generation of Streamlined NTRU Prime. For the encapsulation, we can re-use our polynomial multiplier, the rounding module, the adders, SHA-512 module and multiplexers. The only additional module we would need is a secure module for the fixed-weight sampling of short polynomials. An overview on how this could be done for several schemes including Streamlined NTRU Prime has been presented in [224].

Masking the key generation is a more complex task. It requires two polynomial inversions, where we perform modular multiplications. In addition, the $\mathcal{R}/q$ polynomial input of the $\mathcal{R}/q$ multiplication is not public because it is the inverse of f (see Line 6 in Algorithm 1), which is part of the private key. As a result, we would not be able to use our masked $\mathcal{R}/q$ multiplier, which requires the $\mathcal{R}/q$ input to be public. Both of these aspects mean that a full Boolean masked implementation of the key generation via gadget-based masking is likely infeasible. Alternative approaches to the masked polynomial inversion and key generation for NTRU like schemes have been presented in [225] and [156].

9.4.6. Applicability to Kyber

The efficiency of our gadget-based masking is built upon the fact that the three polynomial multiplications that are carried out each include a secret polynomial with ternary coefficients, where the other one is either ternary and secret as well or has a big coefficient modulus and is public. This enables us to perform schoolbook multiplication in Boolean domain. Notably, Kyber has a similar property: Here, *all* polynomial multiplications have one public input polynomial with "big" coefficients modulo $q = 3\,329$ and one secret input polynomial with "small" coefficients [32].

Moreover, the polynomial degree is far smaller, with 256 compared to 761 for Streamlined NTRU Prime, enabling a faster multiplication. For Kyber, $256^2 = 65\,536$ coefficient additions are to be performed per polynomial multiplication, whereas Streamlined NTRU Prime with $p = 761$ requires $p^2 + p = 579\,882$ coefficient additions. However, Kyber requires more multiplications to be performed, with the number depending on which parameter set is chosen: for $k \in \{2, 3, 4\}$, it requires $k^2 + 2k$ polynomial multiplications, as well as $k^2 + 4k - 1$ polynomial additions, whereas Streamlined NTRU Prime always

requires three polynomial multiplications, one of which only uses "small" coefficients and is thus much cheaper. We compare the cost in terms of the estimated number of coefficient additions in Table 9.8. As seen there, Kyber consistently requires fewer coefficient additions than Streamlined NTRU Prime in the regarding security categories.

Table 9.8.: Number of coefficient additions during decapsulation for Kyber and Streamlined NTRU Prime [20]

Scheme	NIST Category	Core SVP	Polynomial size	Module size	Number of additions	
					big coefficients	small coefficients
Kyber-512	I	118	256	$k = 2$	527 104	0
sntrup653		129	653	—	854 124	427 062
sntrup761	II	153	761	—	1 159 764	579 882
Kyber-768	III	181	256	$k = 3$	988 160	0
sntrup857		175	857	—	1 470 612	735 306
sntrup953	IV	196	953	—	1 818 324	909 162
sntrup1013		209	1013	—	2 054 364	1 027 182
Kyber-1024	V	254	256	$k = 4$	1 580 800	0
sntrup1277		270	1277	—	3 264 012	1 632 006

Another advantage for Kyber is that during key generation, it features no operations that are infeasible to mask in the Boolean domain, which is in contrast to Streamlined NTRU Prime, where this is not possible. The most complex remaining operations in Kyber, both for key generation and decapsulation, are the (de-)compression of coefficients and the sampling for a centered binomial distribution using a Keccak output stream, both of which are feasible in the Boolean domain.

One downside for Kyber is that the secret coefficients have the range of $[-2, 2]$ or $[-3, 3]$. This would require a more complex five-way and seven-way secure multiplexer, which can be constructed out of our Mux2 and Mux3 gadgets, at the cost of additional area and randomness. In addition, a gadget-based masked Kyber implementation would require an NTT core: Kyber requires extending a seed into a public matrix of polynomials, which are assumed to be in the NTT domain. Since the implementation would not perform multiplication in the NTT domain, an inverse transform of each polynomial in the matrix would be required, resulting in k^2 inverse NTTs during decapsulation. Finally, it is noteworthy that the fact that Kyber uses the same polynomial ring for all security levels is no advantage for a gadget-based masked implementation since schoolbook multiplication is used, which also allows for easy parametrization. On the other hand, Streamlined NTRU Prime changes the coefficient modulus over the parameter sets, which does require some manual adjustments. Overall, we leave this as an interesting open idea for future work.

10. Blinding the Re-Encryption of the Fujisaki-Okamoto Transform

This chapter describes multiplicative and additive blinding to the public key and ciphertext for lattice-based PQC KEMs, in order to randomize the otherwise fully deterministic FO transform. As described in Section 5.3, CC-SCAs on the FO transform are particularity powerful, in part due to the fully deterministic nature of the FO transform. Applying blinding increases the difficulty to exploit leakages through side channels during the FO transform due to the additional noise. No higher-level properties are changed, such as the decryption failure probability, and the blinding is fully transparent. While the following description is for Streamlined NTRU Prime, the blinding is compatible both with MLWE schemes such as Kyber and MLWR schemes (Saber), schemes based on the ring version RLWE and RLWR, and finally also schemes based on the NTRU problem, as long as they use the FO transform. The blinding is also compatible with all parameter sets. The full analysis of the protection offered of such blinding is still ongoing work. As a result, we do not yet have a hardware implementation of the FO blinding. Instead, we present a prototype software implementation.

10.1. Blinding preliminaries

In the following we denote a blinded value by a $\sim$, for example, $\tilde{x}$ is a blinded version of x. The variable x can be a scalar or a polynomial. We use the letters m and a to denotes multiplicative and additive blinding values, respectively. In our current context, different kinds of blinding can be applied:

- **Scalar multiplicative blinding.** Generate a random blinding scalar $m \in \mathbb{Z}/q$ with $m^{-1} \in \mathbb{Z}/q$ and compute the blinded variable $\tilde{x} = m \cdot x$, where $\tilde{x}, x \in \mathcal{R}/q$ and $\cdot$ denotes multiplication of a polynomial with a scalar.
- **Polynomial additive blinding.** Generate a random blinding polynomial $a \in \mathcal{R}/q$ and compute the blinded variable $\tilde{x} = a + x$ where $\tilde{x}, x \in \mathcal{R}/q$.
- **Polynomial multiplicative blinding.** Generate a random blinding polynomial $m \in \mathcal{R}/q$ with $m^{-1} \in \mathcal{R}/q$ and compute the blinded variable $\tilde{x} = m \cdot x$, where $\tilde{x}, x \in \mathcal{R}/q$ and $\cdot$ denotes a polynomial multiplication.

10.2. Blinding the public key and ciphertext

With the FO blinding countermeasure presented in this work, the OW-CPA re-encryption is randomized by blinding different intermediates including the public key. This is reminiscent to how the public input of scalar multiplications in ECC are blinded or random-

ized [134]. In addition, the input ciphertext which is used in the comparison in the FO is blinded. This ciphertext comparison is then also modified to account for the blinding. Note that in the description of our blinding scheme the OW-CPA decryption remains unchanged and is not blinded. It is not as sensitive to CC-SCA as the re-encryption is but still naturally requires standard SCA protection like masking. In this section, we mainly focus on the challenges related to the FO transformation, since it is the main target for CC-SCA. The re-encryption offers the most exploitable leakage and hence primarily requires the randomization offered by blinding.

A detailed description of the blinding scheme is given in Algorithm 11. For readability and better comprehension, we use the following colors:

- Green for functions that only operate on public data. They do not require any protection.
- Gray for functions that can be targeted by CC-SCA but cannot be blinded. These functions should be protected by masking them at higher order or using a combination of countermeasures such as masking and shuffling.
- Blue for functions/operations that are modified due to or impacted by the blinding.
- Red for functions/operations that are introduced to either generate the blinding values or blind intermediates.

One important change in the re-encryption is that the rounding (Line 15 in Algorithm 11) and re-encoding of the ciphertext is skipped. This is due to the blinding changing how the polynomials would be rounded. Applying the blinding to other schemes that use rounding (such as Saber) or compression (Kyber) would also require these steps to be skipped. At the end of the re-encryption, the raw, unencoded and unrounded ciphertext polynomial c' is returned. An additional major change is how the ciphertext comparison in Line 16 works. Rather than comparing the encoded ciphertexts (which we do not have because we skipped the rounding and encoding), we subtract the two blinded ciphertext polynomials $\tilde{c}'$ and $\tilde{c}$ from one another, after which we multiply with the inversion of the scalar blind m. This removes both the additive blinding polynomial and the scalar blind. We then check in Line 19 if the resulting polynomial is small, as well as if the two confirmation hashes γ' and γ are identical. Due to the skipping of the rounding, the difference of the $\tilde{c}'$ and $\tilde{c}$ should be in the range of $\{-1, 0, 1\}$. If not, then the received ciphertext C is malformed or invalid. Conceptually, this is similar to the masked range check for Kyber of [152]. There, rather than implementing the masked comparison of the rounded ciphertexts, the authors also skip the rounding. Instead they implement a range check to see if the two ciphertexts are close enough, such that the difference is within the range of that caused by the rounding. Since in Streamlined NTRU Prime the ciphertexts are only rounded to the nearest multiple of the 3, the only possible values are $\{-1, 0, 1\}$.

In Algorithm 11 we show the blinding for a scalar multiplicative blind and a polynomial additive blind. However, a polynomial multiplicative blind can also be used instead of a scalar blind. This has the downside of then requiring a polynomial inversion (Line 16 in Algorithm 11). As we have seen in Section 7.1.4, polynomial inversion is rather expensive in Streamlined NTRU Prime. In contrast, a scalar blind only requires a modular inversion in $\mathbb{Z}/q$, which is much faster. There is also the potential issue of non-invertible multiplicative blinds: In Streamlined NTRU Prime, $\mathbb{Z}/q$ is a field, so the only non-invertible element is the value zero. However, for other schemes that use different rings this may require a more complex blind generation function in order to ensure that the blind is invertible. This is especially the case if a polynomial multiplicative blind is used.

Algorithm 11 Blinded Streamlined NTRU Prime Decapsulation

Input Ciphertext $C \coloneqq (\underline{c}, \gamma)$
Input Private key $S \coloneqq (\underline{k} \coloneqq \mathsf{Encode}(f, g^{-1}),\ \underline{K} \coloneqq \mathsf{Encode}(h),\ \rho,\ \mathsf{hash}_4(\underline{K}))$

1: $c \in \mathcal{R}/q \coloneqq \mathsf{Decode}(\underline{c})$
2: $(f, g^{-1}) \in \mathcal{R}/3 \times \mathcal{R}/3 \coloneqq \mathsf{Decode}(\underline{k})$
3: $h \in \mathcal{R}/q \coloneqq \mathsf{Decode}(\underline{K})$
4: $e \in \mathcal{R}/3 \coloneqq ((3fc) \in \mathcal{R}/q) \bmod 3$ ▷ OW-CPA decryption
5: $r' \in \mathcal{R}/3 \coloneqq g^{-1}e$ ▷ OW-CPA decryption
6: **if** r' does NOT have weight w **then**
7: $\quad r' \coloneqq (1, 1, \ldots, 1, 0, 0, \ldots, 0)$
8: **end if**
9: **repeat**
10: $\quad m \xleftarrow{\$} \mathbb{Z}/q$ ▷ Generate scalar blind
11: **until** $m^{-1} \in \mathbb{Z}/q$
12: $a \xleftarrow{\$} \mathcal{R}/q$ ▷ Generate additive blinds
13: $\tilde{h} \in \mathcal{R}/q \coloneqq hm + a$ ▷ Blind the public key
14: $\tilde{c} \in \mathcal{R}/q \coloneqq cm + r'a$ ▷ Blind the received ciphertext
15: $\tilde{c}' \in \mathcal{R}/q \coloneqq \tilde{h}r'$ ▷ Blinded re-encryption of FO transform
16: $\tilde{d} \in \mathcal{R}/q \coloneqq (\tilde{c}' - \tilde{c})m^{-1}$ ▷ Modified ciphertext comparison
17: $\underline{r}' \coloneqq \mathsf{Encode}(r')$
18: $\gamma' \coloneqq \mathsf{hash}_2(\mathsf{hash}_3(\underline{r}'), \mathsf{hash}_4(\underline{K}))$
19: **if** $\gamma' = \gamma$ and $\tilde{d} \in \mathsf{small}$ **then** ▷ Modified ciphertext comparison
20: $\quad ss \coloneqq \mathsf{hash}_1(\mathsf{hash}_3(\underline{r}'), C)$
21: $\quad$ **return** ss
22: **else**
23: $\quad ss' \coloneqq \mathsf{hash}_0(\mathsf{hash}_3(\rho), C)$
24: $\quad$ **return** ss'
25: **end if**

10.3. Analysis

Due to the blinding, the public key input to the FO transform is now randomized for every decapsulation, and unknown to an attacker. As a result, side channel attacks targeting the re-encryption of the sensitive plaintext (Line 15 in Algorithm 11) can now no longer rely on knowing the public key, as it is blinded. Likewise, an attacker targeting the ciphertext comparison (Line 16 in Algorithm 11) can no longer rely on knowing the ciphertext, as it is also blinded. Additional traces and measurements would be needed to first deduce the blinding scalars and blinding polynomials. Also, due to the randomization, it is more difficult for an attacker to build templates of a target device. At the same time, because the CPA decryption is not protected, the blinding should always be implemented together with additional side channel countermeasures such as masking. The same also applies to the encoding of r' as well as to all calls to the SHA-512 hash function. In addition, the RNG that creates the blinding elements should also be protected (e.g. a masked Keccak implementation). However, the exact impact on the side-channel resistance by the blinding is still an ongoing work. In particular, whether the blinding would allow us to reduce the masking degree is still an open question.

10.3.1. Performance

A major benefit of the blinding is that it is very lightweight, especially when compared to masking. The overheads introduced by the different blinding techniques are shown in Table 10.1. We can see that the overhead for the scalar multiplicative blinding is very low. The polynomial additive blinding is slightly more expensive, as it requires an additional $\mathcal{R}/q \cdot \mathcal{R}/3$ polynomial multiplication, as well as the much higher randomness needed to sample an $\mathcal{R}/q$ polynomial. An optional speedup here would be to reduce the coefficient range of the additive blind: Rather than sampling each polynomial coefficient from $\mathbb{Z}/q$, they could for example only have a range of 8 bit. This would reduce the amount of randomness needed.

Table 10.1.: The additional operations and overhead needed to implement the different FO blinding schemes for Streamlined NTRU Prime and the sntrup761 parameter set.

Blinding Scheme	RNG (bits)	$\mathcal{R}/q \cdot \mathbb{Z}/q$	$\mathcal{R}/q \cdot \mathcal{R}/3$	$\mathcal{R}/q \pm \mathcal{R}/q$	Inversion
Scalar Mult.	13	3	0	1	1 in $\mathbb{Z}/q$
Poly. Mult.	9893	0	4	1	1 in $\mathcal{R}/q$
Poly. Additive	9893	0	1	3	0
Scalar Mult. and Poly. Additive	9906	3	1	3	1 in $\mathbb{Z}/q$

Nevertheless, both techniques are still much more lightweight than for example masking. In [172] the authors describe that a very high masking degree is needed in order to properly defend against CC-SCA. This in turn leads to a significant overhead, which

can be seen in the benchmarking of high-order masked implementations such as in [173] and [152]. To be specific, protecting against CC-SCA requires two additional shares in low-noise settings (e.g., a pure software implementation), and one additional share in a high-noise setting [172, 226] (e.g., a hardware implementation). If we compare the overhead of our masked Streamlined NTRU Prime implementation in Table 9.7, increasing the masking degree from one to two causes an 85.6 % increase in area, a 200 % increase in randomness and a 25.9 % increase in latency. Should the blinding allow implementations to decrease the masking degree, then this would lead to a significant performance improvement.

10.3.2. Prototype Software Implementation

We verify the functionality by implementing the scalar multiplicative and polynomial additive blinding in the Streamlined NTRU Prime C reference implementation as well as in the SageMath reference implementation. SageMath is a computer algebra system based on Python, with additional higher-level math libraries [227]. The C reference implementation is constant-time, but otherwise offers no protection against side-channel attacks [9]. It also does not use highly-optimized code, instead targeting clarity. As a result, core functions such as polynomial multiplication are not particularly fast. We benchmark our C implementation using the SUPERCOP framework [191] on an Intel i5-8350U processor. The results can be found in Table 10.2

Table 10.2.: Benchmarking of the blinding schemes applied to the Streamlined NTRU Prime decapsulation for the sntrup761 parameter set on a Intel i5-8350U processor. We list the 25th, 50th and 75th percentile cycle counts, as well as the number of calls to the RNG and the total number of random bytes requested.

Blinding Scheme	Random Bytes	RNG Calls	Cycle count		
			25 %	50 %	75 %
Ref [9]	0	0	40614418	40699910	40783502
Scalar Mult.	4	1	40706446	40803753	40862309
Scalar Mult. and Poly. Additive	3048	762	54195653	54405545	54832069

We can see that when only the scalar multiplicative blinding is applied, the cycle count is almost identical to the reference implementation without any blinding. This is due to the fact that we do not perform the ciphertext rounding and encoding. This compensates the overhead of the scalar multiplication, polynomial additions and modular inversion. When we add the polynomial additive blinding, we have a noticeable increase of the cycle counts. This is due to the additional $\mathcal{R}/q \cdot \mathcal{R}/3$ multiplication. Overall, the total overhead is roughly 33 %, which corresponds to the increase of polynomial multiplications from 3 to 4. We can thus conjecture that the additional RNG calls currently only have a minor impact on the cycle count. However, this is likely to change if more optimized polynomial multiplication routines were to be used.

Part IV.

Conclusion & Final Remarks

11. Conclusion

We have presented several complete hardware implementations of Streamlined NTRU Prime. This includes both low-area designs, optimized for minimal area usage, as well as high-speed designs which are optimized for maximum performance. For these designs, we have introduced several techniques to improve the design, such as batch inversion and highly efficient schoolbook polynomial multipliers. Our implementation results show that our designs outperform other NTRU hardware implementations in nearly all aspects, and are even competitive in certain areas with state-of-the-art hardware implementations of Kyber and Saber. Notably, our high-speed implementation has an encapsulation latency of just 5.63 µs for the sntrup653 parameter set, which places it as one of the fastest encapsulations of *any* PQC algorithms in the literature. In addition, our use of batch inversion successfully mitigates the otherwise very slow polynomial inversion inside the key generation, allowing our Streamlined NTRU Prime key generation to outperform the key generation of NTRU. These benchmark feats are remarkable, as Streamlined NTRU Prime has been criticized for its slow performance [31]. However, recall that the initial main design goal of Streamlined NTRU Prime was to reduce the potential attack surface without sacrificing too much speed [7]. Our implementations show that this goal was successful, and that these design choices of Streamlined NTRU Prime are not an obstacle to highly efficient and competitive implementations. In fact, some of the design choices, such as the use of small polynomials, are what allows our implementations to achieve their high performance in the first place. Nevertheless, there are still some areas that could be improved further: We identified some of the bottlenecks in our design, which are primarily the SHA-512 module, $\mathcal{R}/q$ decoder module and the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier. Additional optimizations of these parts could improve performance even further.

Although this thesis has focused on Streamlined NTRU Prime, many of our modules can be use for other cryptographic schemes, such as NTRU LPRime and NTRU with little to no modifications. Modules such as polynomial multiplication, SHA-512 and modular reductions are a core of many cryptographic algorithms, and our modules are highly suitable for many different algorithms. Other modules, such as the sorting and batch inversion modules, are also of interest in areas beyond cryptography, as these operations are widespread in computer science. In addition, we also show that the schoolbook multiplication algorithm is the most efficient multiplication algorithm for both high-speed and low-area hardware implementations of Streamlined NTRU Prime on both FPGAs and ASICs, and not the asymptotically faster NTT. This is in line with results for Saber such as in [88], where schoolbook multipliers were also more efficient than NTT-based ones. In both cases, this is possible due to the use of "small" polynomials, where at least one polynomial has coefficients that are only a few bits large and allow us to perform a polynomial multiplication without actually requiring a proper multiplier circuit. As a result, we conclude that schoolbook multipliers are an efficient

option for structured-lattice cryptography, as long as the cryptosystem uses "small" polynomials. This in turn means that these cryptosystems are not required to use NTT-friendly rings in order to enjoy efficient hardware implementations. An interesting future work to further investigate this would be a full ASIC synthesis of our Streamlined NTRU Prime implementations.

We have also presented the first gadget-based masked implementation of any public key cryptographic scheme. Our design relies solely on Boolean masking, and it is competitive regarding area demand to other protected PQC implementations while still offering reasonable latency. Because we only use Boolean masking, we do not need to perform any masking conversions, which have been an attack vector on protected PQC implementations [128, 129]. We are also able to use formal verification to verify the side-channel security of our modules, giving us stronger guarantees in comparison to relying purely on practical evaluation. An additional advantage is the support of an arbitrary masking degree. To our knowledge, our design is the first higher-order masked hardware implementation of a PQC scheme in the literature. For the first-order secure instance of the implementation, 19 923 LUTs, 19 725 FFs, and 8.5 BRAMs are utilized, reaching a frequency of 200 MHz on an Artix-7 FPGA. Implemented as an ASIC, the first-order secure instance consumes 201k GE and 189 kbit of SRAM, reaching a frequency of 207 MHz. This results in a latency of only 9.35 ms on an FPGA and 9.03 ms as an ASIC, with a peak demand of fresh randomness of 310 bit per clock cycle. Like with the unmasked implementation, our masked design shows that the design choices of Streamlined NTRU Prime are no obstacle to an efficient masked implementation. Nevertheless, further optimization of the hashing module, including a potential switch to a masked Keccak, could significantly reduce the area and randomness consumption. At the same time, additional adders in the polynomial multiplier could drastically reduce the latency. While we currently have only implemented the decapsulation, many of our masked modules can be reused to implement the encapsulation as well: The only missing piece is a secure module to generate short polynomials. Looking beyond Streamlined NTRU Prime, we also analyzed the applicability of our concept to the designated NIST standard algorithm Kyber, finding that gadget-based masking could be efficient for Kyber as well. In addition, we also present the first arbitrary-order masked SHA-512 implementation in the open literature as part of our design. While protected SHA-512 implementation are available commercially in devices such as smartcards [222], to our knowledge no designs have been published in academic journals or conferences. Because SHA-512 is a widely deployed hash algorithm, such as in the ECC signature scheme EdDSA [228], a side-channel protected implementation of SHA-512 is of general interest.

As part of the work on masked implementations, we presented a case study of applying the automated masking tool AGEMA [22] to a PKE scheme. Although our study shows that it is theoretically possible, we conclude that the process is currently far from optimal, as we not only need a non-trivial amount of manual modifications, but the resulting masked design is also highly inefficient. The inefficiency is a result of design, architecture

and ASIC synthesis choices that, while suitable for unmasked designs, cause a significant overhead in a masked design. However, we also show that modifications to ASIC EDA tools could make this approach much more feasible in the future.

Finally, we introduced a new side-channel protection measure in the form of the Fujisaki-Okamoto transform blinding. While the analysis of the blinding is not yet complete, it has the potential of protecting PQC schemes from particularly powerful CC-SCA at comparatively low cost. In our software benchmarks, the blinding only adds roughly 33 % overhead, which is significantly cheaper than the overhead of increasing the masking degree. The blinding can not only be applied to Streamlined NTRU Prime, but also to any lattice scheme that uses the FO transform, including now-standardized Kyber.

11.1. Outlook

Despite not being selected for standardization by NIST, NTRU Prime has already received attention and production deployments, particularly from open-source projects [55, 56]. Our hardware implementations are suitable in these use-cases, as we support a wide range of optimization targets: From low-area to high-speed variants, as well as side-channel protected versions. Our implementations can be used as accelerators and co-processors, in order to increase performance, lower power consumption or protect against side-channel attacks. Such benefits can help mitigate the overhead induced by the transition to PQC.

In addition, the fact that our implementations outperform NTRU hardware implementations and are also competitive in certain areas with Kyber can help system engineers in deciding which PQC algorithm to deploy: NTRU has been promoted as a more conservative option over Kyber, as NTRU predates Kyber by over a decade [31, 229]. Streamlined NTRU Prime then further improves the security over NTRU by reducing the attack surface and mitigating a number of potential security risks [7, 51]. Because of the high efficiency and performance of our hardware implementations, engineers who wish to use a more conservative cryptosystem can now do so in the form of Streamlined NTRU Prime without sacrificing performance. In fact, by using our hardware implementation, we actually *gain* performance over NTRU. This also enables alternatives to other conservative algorithms such as Classic McEliece [74, 75] or Frodo [76, 77] that incur a much larger performance penalty [31].

A. Appendix

A.1. Precomputed Parameters for the En- & Decoder

Table A.1.: Round information for sntrup857 $\mathcal{R}/q$-encode.

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	857	5167	2	856	5167	N/A
2	429	408	1	214	5167	N/A
3	215	651	1	107	5167	N/A
4	108	1656	1	53	5167	2
5	54	10713	2	52	131	1
6	27	1752	1	13	5483	N/A
7	14	11991	2	12	5483	2
8	7	2194	2	6	1004	N/A
9	4	74	0	0	1004	1
10	2	5476	N/A	N/A	291	1
11	1	N/A	N/A	N/A	6225	2

Table A.2.: Round information for sntrup857 rounded-encode.

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	857	1723	1	428	1723	N/A
2	429	11597	2	428	1723	N/A
3	215	2053	2	214	1723	N/A
4	108	65	0	0	1723	1
5	54	4225	2	52	438	1
6	27	273	1	13	7229	N/A
7	14	292	1	6	7229	1
8	7	334	1	3	8246	N/A
9	4	436	1	1	8246	1
10	2	743	N/A	N/A	14044	2
11	1	N/A	N/A	N/A	160	1

Table A.3.: Round information for sntrup653 $\mathcal{R}/q$-encode.

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	653	4621	2	652	4621	N/A
2	327	326	1	163	4621	N/A
3	164	416	1	81	4621	1
4	82	676	1	40	7510	2
5	41	1786	1	20	78	N/A
6	21	12461	2	20	78	N/A
7	11	2370	2	10	78	N/A
8	6	86	0	0	0	0
9	3	7396	2	2	6708	N/A
10	2	835	N/A	N/A	6708	2
11	1	N/A	N/A	N/A	86	1

Table A.4.: Round information for sntrup653 rounded-encode.

Round	len(M)	m_0	regular output	subtotal	m_1	last output
1	653	1541	1	326	1541	N/A
2	327	9277	2	326	1541	N/A
3	164	1314	1	81	1541	1
4	82	6745	2	80	7910	2
5	41	695	1	20	815	N/A
6	21	1887	1	10	815	N/A
7	11	13910	2	10	815	N/A
8	6	2953	2	4	815	1
9	3	134	0	1	9402	N/A
10	2	71	N/A	N/A	9402	1
11	1	N/A	N/A	N/A	2608	2

A.2. C Code of Polynomial Inversion

```
/* out = 1/(3*in) in Rq. Returns 0 if recip succeeded; else -1
   */
static int Rq_recip3(Fq *out,const small *in) {
  Fq f[p+1],g[p+1],v[p+1],r[p+1],scale,f0,g0;
  int i,loop,delta,swap,t;

  for (i = 0;i < p+1;++i) v[i] = 0;
  for (i = 0;i < p+1;++i) r[i] = 0;
  for (i = 0;i < p;++i) f[i] = 0;
```

```
  for (i = 0;i < p;++i) g[p-1-i] = in[i];
  delta = 1; g[p] = 0; r[0] = 1; f[0] = 1; f[p-1] = f[p] = -1;

  for (loop = 0;loop < 2*p-1;++loop) {
    for (i = p;i > 0;--i) v[i] = v[i-1];
    v[0] = 0;
    // negative_mask returns -1 if input < 0; else 0
    // nonzero_mask returns -1 if input != 0; else 0
    swap = negative_mask(-delta) & nonzero_mask(g[0]);
    delta ^= swap&(delta^-delta);
    delta += 1;

    for (i = 0;i < p+1;++i) {
      t = swap&(f[i]^g[i]); f[i] ^= t; g[i] ^= t;
      t = swap&(v[i]^r[i]); v[i] ^= t; r[i] ^= t;
    }

    f0 = f[0]; g0 = g[0];
    // Fq_freeze computes the input mod q
    for (i = 0;i < p+1;++i) g[i] = Fq_freeze(f0*g[i]-g0*f[i]);
    for (i = 0;i < p+1;++i) r[i] = Fq_freeze(f0*r[i]-g0*v[i]);

    for (i = 0;i < p;++i) g[i] = g[i+1];
    g[p] = 0;
  }
  // Fq_recip computes the modular inverse of the input
  scale = Fq_recip(f[0]);
  for (i = 0;i < p;++i) out[i] = Fq_freeze(scale*(int32)v[p-1-i
   ]);

  return nonzero_mask(delta);
}
```

Listing A.1: The C code of the polynomial inversion algorithm, from the Streamlined NTRU Prime reference implementation [9].

Bibliography

[1] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, November 1994. [Cited on pages 1 and 9.]

[2] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM Symposium on Theory of Computing*, pages 212–219. ACM Press, May 1996. [Cited on pages 1 and 9.]

[3] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography*. Springer, Berlin, Germany, 2009. [Cited on pages 1, 9, 10, and 11.]

[4] Michele Mosca. Cybersecurity in a Quantum World: Will we be ready? `https://csrc.nist.gov/csrc/media/events/workshop-on-cybersecurity-in-a-post-quantum-world/documents/presentations/session8-mosca-michele.pdf`, 2015. [Online; accessed 31-01-2024]. [Cited on pages xvii, 1, and 9.]

[5] PQCRYPTO. Post-quantum cryptography for long-term security. `https://pqcrypto.eu.org/index.html`, 2015. [Online; accessed 31-01-2024]. [Cited on page 1.]

[6] NIST. NIST post-quantum cryptography standardization. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`. [Online; accessed 31-01-2024]. [Cited on pages 1 and 10.]

[7] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: Reducing Attack Surface at Low Cost. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography 2017*, volume 10719 of *LNCS*, pages 235–260, August 2017. [Cited on pages xxvii, 2, 12, 13, 15, 16, 17, 25, 57, 58, 153, and 155.]

[8] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, pages 267–288. Springer, 2006. [Cited on pages 2, 12, 14, and 15.]

[9] 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. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. [Cited on pages xxi, xxiii, xxvii, 2, 12, 13, 14, 15, 16, 17, 23, 24, 33, 34, 51, 57, 58, 59, 70, 90, 99, 119, 149, and 159.]

[10] Adrian Marotzke. A Constant-Time Full Hardware Implementation of Streamlined NTRU Prime. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020. [Cited on pages xvii, 3, 4, 23, 49, 50, 57, 59, 63, 71, 75, 77, 78, 81, 82, 83, 85, 86, 87, 88, 89, 90, 91, 95, 96, and 114.]

[11] Adrian Marotzke. A Constant-Time Full Hardware Implementation of Streamlined NTRU Prime - CARDIS 2020. `https://www.youtube.com/watch?v=y_tH-Go_3cI`, 2020. [Online; accessed 31-01-2024]. [Cited on page 3.]

[12] Adrian Marotzke. A Constant-Time Hardware Implementation of Streamlined NTRU Prime. `https://github.com/AdrianMarotzke/SNTRUP`, 2020. [Online; accessed 31-01-2024]. [Cited on pages 3, 50, 83, 90, and 91.]

[13] 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: round-3 updates. `https://csrc.nist.gov/CSRC/media/Presentations/ntruprime-round-3-presentation/images-media/session-6-ntruprime-bernstein.pdf`, 2021. [Online; accessed 31-01-2024]. [Cited on page 3.]

[14] NIST. Third PQC Standardization Conference. `https://csrc.nist.gov/Events/2021/third-pqc-standardization-conference`, 2021. [Online; accessed 31-01-2024]. [Cited on page 3.]

[15] Grishma R Pandeya, Tuğrul U Daim, and Adrian Marotzke. A strategy roadmap for post-quantum cryptography. *Roadmapping Future: Technologies, Products and Services*, pages 171–207, 2021. [Cited on page 3.]

[16] Adrian Marotzke. PQC in the industry: The risks and challenges, 2022. `https://www.uni-jena.de/forschung/forschungsprofil/profillinie-liberty/freiheitsraeume-und-freiheitssicherung-im-digitalen-staat/quantum-projekt/security-in-the-quantum-age/program` [Accessed 07.10.2021]. [Cited on page 3.]

[17] Bo-Yuan Peng, Adrian Marotzke, Ming-Han Tsai, Bo-Yin Yang, and Ho-Lin Chen. Streamlined NTRU Prime on FPGA. *Journal of Cryptographic Engineering*, pages 1–20, 2022. [Cited on pages xvii, xxiii, 3, 4, 23, 27, 29, 32, 33, 49, 51, 53, 54, 55, 57, 59, 60, 61, 62, 63, 65, 66, 68, 71, 72, 73, 74, 75, 76, 77, 78, 81, 82, 83, 85, 86, 87, 88, 89, 90, 91, 92, 93, 95, 96, 105, 114, 120, 125, and 137.]

[18] Adrian Marotzke. Streamlined NTRU Prime on FPGA. `https://github.com/AdrianMarotzke/SNTRUP_on_FPGA`, 2021. [Online; accessed 31-01-2024]. [Cited on page 3.]

[19] Adrian Marotzke. To NTT or not to NTT: Polynomial multiplication strategies for hardware implementations of lattice cryptography. https://events.btq.li/pqc-workshop-2022/, 2022. [Online; accessed 31-01-2024]. [Cited on page 3.]

[20] Georg Land, Adrian Marotzke, Jan Richter-Brockmann, and Tim Güneysu. Gadget-based Masking of Streamlined NTRU Prime Decapsulation in Hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):1–26, 2023. https://tches.iacr.org/index.php/TCHES/article/view/11238. [Cited on pages xix, xxv, 4, 5, 69, 113, 120, 123, 125, 129, 131, 137, 138, 139, 141, and 143.]

[21] Adrian Marotzke. Masked-SNTRUP. https://github.com/AdrianMarotzke/Masked-SNTRUP, 2023. [Online; accessed 31-01-2024]. [Cited on pages 4 and 120.]

[22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):589–629, 2022. [Cited on pages 4, 44, 125, 138, 139, and 154.]

[23] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. [Cited on page 9.]

[24] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426, August 1986. [Cited on page 9.]

[25] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987. [Cited on page 9.]

[26] J. P. Buhler, H. W. Lenstra, and Carl Pomerance. Factoring integers with the number field sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, pages 50–94, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. [Cited on page 9.]

[27] John M Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978. [Cited on page 9.]

[28] Junpei Yamaguchi, Masafumi Yamazaki, Akihiro Tabuchi, Takumi Honda, Tetsuya Izu, and Noboru Kunihiro. Estimation of Shor's Circuit for 2048-bit Integers based on Quantum Simulator. Cryptology ePrint Archive, Report 2023/092, 2023. [Cited on page 9.]

[29] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, April 2021. [Cited on page 9.]

[30] John Timmer. IBM pushes qubit count over 400 with new processor. https://arstechnica.com/science/2022/11/ibm-pushes-qubit-count-over-400-with-new-processor/. [Online; accessed 31-01-2024]. [Cited on page 9.]

[31] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022. [Cited on pages 10, 13, 99, 153, and 155.]

[32] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`. [Cited on pages 10, 13, 27, 84, 95, 98, and 142.]

[33] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`. [Cited on page 10.]

[34] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`. [Cited on page 10.]

[35] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`. [Cited on page 10.]

[36] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 203. August 13, 2024, U.S. Department of Commerce, Washington, D.C., 2024. [Cited on pages 10 and 13.]

[37] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 204. August 13, 2024, U.S. Department of Commerce, Washington, D.C., 2024. [Cited on page 10.]

[38] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 205. August 13, 2024, U.S. Department of Commerce, Washington, D.C., 2024. [Cited on page 10.]

[39] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982. [Cited on page 11.]

[40] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66:181–199, 1994. [Cited on page 11.]

[41] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *33rd ACM Symposium on Theory of Computing*, pages 601–610. ACM Press, July 2001. [Cited on page 11.]

[42] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20, December 2011. [Cited on page 11.]

[43] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *28th ACM Symposium on Theory of Computing*, pages 99–108. ACM Press, May 1996. [Cited on page 12.]

[44] Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 52–61, May 1997. [Cited on pages 12 and 14.]

[45] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of NTRU encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 226–246, August 2003. [Cited on pages 12 and 17.]

[46] Jeff Hoffstein, Nick Howgrave-Graham, Jill Pipher, and William Whyte. Practical lattice-based cryptography: NTRUEncrypt and NTRUSign. In *The LLL Algorithm: Survey and Applications*, pages 349–390. Springer, 2009. [Cited on pages 12 and 14.]

[47] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM Symposium on Theory of Computing*, pages 84–93. ACM Press, May 2005. [Cited on page 12.]

[48] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM Symposium on Theory of Computing*, pages 169–178. ACM Press, May / June 2009. [Cited on page 12.]

[49] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635, December 2009. [Cited on page 12.]

[50] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23, May / June 2010. [Cited on page 12.]

[51] NTRU Prime Risk-Management Team. Risks of lattice KEMs. `https://ntruprime.cr.yp.to/warnings.html`, 2021. [Online; accessed 31-01-2024]. [Cited on pages 12, 17, 99, and 155.]

[52] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions`. [Cited on page 12.]

[53] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime. Technical report, National Institute of Standards and Technology, 2019. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions`. [Cited on page 12.]

[54] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: International Conference on Theory and Practice of Public-Key Cryptography*, volume 3958 of *LNCS*, pages 207–228, April 2006. [Cited on page 13.]

[55] OpenSSH. OpenSSH 9.0 release notes. `https://www.openssh.com/txt/release-9.0`. [Online; accessed 31-01-2024]. [Cited on pages 13 and 155.]

[56] OpenBSD. Add experimental post-quantum hybrid key exchange method. `https://github.com/openbsd/src/commit/9b50bc253d6cf270fe0a001333c163c7cd5422e5`. [Online; accessed 31-01-2024]. [Cited on pages 13 and 155.]

[57] Simon Josefsson. Streamlined NTRU Prime: sntrup761. Internet-Draft draft-josefsson-ntruprime-streamlined-00, Internet Engineering Task Force, May 2023. Work in Progress. [Cited on page 13.]

[58] Simon Josefsson. Hybrid Streamlined NTRU Prime sntrup761 and X25519 with SHA-512: sntrup761+x25519+sha512. Internet-Draft draft-josefsson-ntruprime-hybrid-00, Internet Engineering Task Force, May 2023. Work in Progress. [Cited on page 13.]

[59] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016. Document ID: 0462d84a3d34b12b75e8f5e4ca032869, `http://cryptojedi.org/papers/#newhope`. [Cited on pages xxiii, 14, and 99.]

[60] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 27–47, May 2011. [Cited on page 14.]

[61] Alice Pellet-Mary and Damien Stehlé. On the hardness of the NTRU problem. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 3–35, December 2021. [Cited on pages 14 and 15.]

[62] Nick Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 150–169, August 2007. [Cited on page 14.]

[63] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, William Whyte, and Zhenfei Zhang. Choosing parameters for NTRUEncrypt. In Helena Handschuh, editor, *CT-RSA 2017: The Cryptographer's Track at the RSA Conference*, volume 10159 of *LNCS*, pages 3–18, February 2017. [Cited on page 14.]

[64] Daniele Micciancio and Michael Walter. Practical, predictable lattice basis reduction. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 820–849, May 2016. [Cited on page 16.]

[65] Daniel J. Bernstein and Tanja Lange. Non-randomness of S-unit lattices. Cryptology ePrint Archive, Report 2021/1428, 2021. [Cited on page 17.]

[66] Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 559–585, May 2016. [Cited on page 17.]

[67] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In Robert Krauthgamer, editor, *27th Symposium on Discrete Algorithms*, pages 893–902. ACM-SIAM, January 2016. [Cited on page 17.]

[68] Qian Guo, Thomas Johansson, and Jing Yang. A novel CCA attack using decryption errors against LAC. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 82–111, December 2019. [Cited on page 17.]

[69] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: International Conference on Theory and Practice of Public-Key Cryptography, Part II*, volume 11443 of *LNCS*, pages 565–598, April 2019. [Cited on page 17.]

[70] Daniel J. Bernstein, Leon Groot Bruinderink, Tanja Lange, and Lorenz Panny. HILA5 Pindakaas: On the CCA security of lattice-based encryption with error

correction. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18*, volume 10831 of *LNCS*, pages 203–216, May 2018. [Cited on page 17.]

[71] Chris Hall, Ian Goldberg, and Bruce Schneier. Reaction attacks against several public-key cryptosystems. In Vijay Varadharajan and Yi Mu, editors, *ICICS 99: International Conference on Information and Communications Security*, volume 1726 of *LNCS*, pages 2–12, November 1999. [Cited on page 17.]

[72] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554, August 1999. [Cited on page 17.]

[73] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011. [Cited on page 17.]

[74] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. The deep space network progress report 42-44, Jet Propulsion Laboratory, California Institute of Technology, January/February 1978. `https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF`. [Cited on pages 17 and 155.]

[75] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions`. [Cited on pages 17 and 155.]

[76] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. [Cited on pages 17 and 155.]

[77] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: Conference on Computer and Communications Security*, pages 1006–1018. ACM Press, October 2016. [Cited on pages 17 and 155.]

[78] Apon, Daniel. NIST assignments of platforms on implementation efforts to PQC teams. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJx`

Mq0_90gU/m/qbGEs3TXGwAJ. [Online 7-February-2019; accessed 15-October-2021]. [Cited on page 19.]

[79] Xilinx, Inc. *UG574: UltraScale Architecture Configurable Logic Block*, 1.5 edition, February 2017. https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf. [Cited on pages 19, 20, and 76.]

[80] Xilinx, Inc. *UG474: 7 Series FPGAs Configurable Logic Block*, 1.8 edition, September 2016. https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB. [Cited on pages 19, 20, and 76.]

[81] Xilinx, Inc. *UG579: UltraScale Architecture DSP Slice*, 1.11 edition, August 2021. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf. [Cited on page 20.]

[82] Xilinx, Inc. *UG479: 7 Series DSP48E1 Slice*, 1.10 edition, March 2018. https://www.xilinx.com/content/dam/xilinx/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf. [Cited on page 20.]

[83] Xilinx, Inc. *UG573: UltraScale Architecture Memory Resources*, 1.13 edition, September 2021. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf. [Cited on page 20.]

[84] Xilinx, Inc. *UG473: 7 Series FPGAs Memory Resources*, 1.14 edition, July 2019. https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf. [Cited on page 20.]

[85] Michael John Sebastian Smith. *Application-specific Integrated Circuits*. Addison-Wesley VLSI systems series. Addison-Wesley, 1997. [Cited on pages 20 and 21.]

[86] Silvaco. 45nm Nangate open cell library. https://si2.org/open-cell-library/. [Online; accessed 31-01-2024]. [Cited on pages xxiii, xxv, 21, 85, 125, 129, and 132.]

[87] 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. [Cited on pages xxiii, 23, 24, 59, 87, and 88.]

[88] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-speed hardware architectures and FPGA benchmarking of Crystals-Kyber, NTRU, and SABER. *IEEE Transactions on Computers*, 2022. [Cited on pages xviii, 23, 25, 26, 27, 49, 50, 57, 81, 87, 88, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 107, and 153.]

[89] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: Improved polynomial multiplication and inversion in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):557–588, 2022. [Cited on page 23.]

[90] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 845–862. USENIX Association, August 2022. [Cited on pages 23, 25, and 59.]

[91] Peter L Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987. [Cited on pages xxi and 25.]

[92] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 459–474, September / October 2011. [Cited on pages xxi, 25, 26, and 49.]

[93] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, 2020. [Cited on pages xxi, 25, 26, 49, and 50.]

[94] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire O'Neill. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2459–2463, 2019. [Cited on page 25.]

[95] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962. [Cited on pages xxi, 25, and 26.]

[96] Stephen A Cook and Stal̇ O Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969. [Cited on page 26.]

[97] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based kem. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. [Cited on page 26.]

[98] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(3):1146–1159, 2021. [Cited on page 26.]

[99] R. Agarwal and C. 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. [Cited on page 26.]

[100] R.C. Agarwal and C.S. Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975. [Cited on page 26.]

[101] 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. [Cited on page 26.]

[102] Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158, October 2012. [Cited on pages 27 and 29.]

[103] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16: Cryptology and Network Security*, volume 10052 of *LNCS*, pages 124–139, November 2016. [Cited on pages xxi, 27, 28, and 29.]

[104] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of NewHope-NIST. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):49–72, 2020. [Cited on pages 27, 54, 63, and 64.]

[105] 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. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):217–238, 2021. [Cited on pages 27, 29, and 114.]

[106] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. [Cited on pages 27, 29, 30, 31, and 114.]

[107] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 209–228, July 2019. [Cited on page 27.]

[108] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, 71(11):2829–2843, 2020. [Cited on pages xxi and 28.]

[109] Irving J Good. Random motion on a finite abelian group. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 47, pages 756–762. Cambridge University Press, 1951. [Cited on pages 29 and 30.]

[110] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in context*, pages 1–18, 2008. [Cited on pages xvii, 33, 35, and 36.]

[111] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6):65–68, 2009. [Cited on page 37.]

[112] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA 2006: The Cryptographer's Track at the RSA Conference*, volume 3860 of *LNCS*, pages 1–20, February 2006. [Cited on page 37.]

[113] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security 2003*. USENIX Association, August 2003. [Cited on page 37.]

[114] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 359–386, August 2020. [Cited on page 37.]

[115] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397, August 1999. [Cited on page 37.]

[116] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002: Conference on Cryptographic Hardware and Embedded Systems*, volume 2523 of *LNCS*, pages 29–45, August 2003. [Cited on page 37.]

[117] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008. [Cited on page 37.]

[118] Zhuang Xu, Owen Michael Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. *IEEE Transactions on Computers*, 2021. [Cited on page 37.]

[119] Bo-Yeon Sim, Aesun Park, and Dong-Guk Han. Chosen-ciphertext clustering attack on CRYSTALS-KYBER using the side-channel leakage of barrett reduction. *IEEE Internet Things Journal*, 9(21):21382–21397, 2022. [Cited on page 37.]

[120] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure Kyber. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):88–113, 2021. [Cited on page 37.]

[121] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. Fault-Enabled Chosen-Ciphertext Attacks on Kyber. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2021. [Cited on page 37.]

[122] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked IND-CCA secure Saber KEM implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):676–707, 2021. [Cited on pages 37 and 43.]

[123] Emre Karabulut and Aydin Aysu. FALCON Down: Breaking FALCON Post-Quantum Signature Scheme through Side-Channel Attacks. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 691–696. IEEE, 2021. [Cited on page 37.]

[124] Amund Askeland and Sondre Rønjom. A side-channel assisted attack on NTRU. Cryptology ePrint Archive, Report 2021/790, 2021. [Cited on page 37.]

[125] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, 2020. [Cited on pages 37 and 43.]

[126] Elena Dubrova, Kalle Ngo, and Joel Gärtner. Breaking a fifth-order masked implementation of CRYSTALS-Kyber by copy-paste. Cryptology ePrint Archive, Report 2022/1713, 2022. [Cited on page 37.]

[127] Jian Wang, Weiqiong Cao, Hua Chen, and Haoyuan Li. Practical side-channel attack on masked message encoding in latticed-based KEM. Cryptology ePrint Archive, Report 2022/859, 2022. [Cited on page 37.]

[128] Kalle Ngo, Elena Dubrova, and Thomas Johansson. Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis. In Chip-Hong Chang, Ulrich Rührmair, Stefan Katzenbeisser, and Debdeep Mukhopadhyay, editors, *ASHES@CCS 2021: Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security, Virtual Event, Republic of Korea, 19 November 2021*, pages 51–61. ACM, 2021. [Cited on pages 37, 114, 139, and 154.]

[129] Kalle Ngo, Ruize Wang, Elena Dubrova, and Nils Paulsrud. Side-channel attacks on lattice-based KEMs are not prevented by higher-order masking. Cryptology ePrint Archive, Report 2022/919, 2022. [Cited on pages 37, 114, 139, and 154.]

[130] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-Trace Side-Channel Attacks on ω-Small Polynomial Sampling: With Applications to NTRU, NTRU

Prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pages 35–45. IEEE, 2021. [Cited on page 37.]

[131] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002: Conference on Cryptographic Hardware and Embedded Systems*, volume 2523 of *LNCS*, pages 13–28, August 2003. [Cited on page 37.]

[132] Prasanna Ravi, Martianus Frederic Ezerman, Shivam Bhasin, Anupam Chattopadhyay, and Sujoy Sinha Roy. Will you cross the threshold for me? Generic side-channel assisted chosen-ciphertext attacks on NTRU-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):722–761, 2022. [Cited on page 37.]

[133] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113, August 1996. [Cited on page 38.]

[134] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *CHES'99: Conference on Cryptographic Hardware and Embedded Systems*, volume 1717 of *LNCS*, pages 292–302, August 1999. [Cited on pages 38 and 146.]

[135] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures - engineering a side-channel resistant post-quantum signature scheme with compact signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, April 2018. [Cited on page 38.]

[136] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-LWE masking. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 233–244, 2016. [Cited on page 38.]

[137] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015: Conference on Cryptographic Hardware and Embedded Systems*, volume 9293 of *LNCS*, pages 683–702, September 2015. [Cited on page 38.]

[138] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. [Cited on page 38.]

[139] Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In Bruce Schneier, editor, *FSE 2000: Fast Software Encryption Conference*, volume 1978 of *LNCS*, pages 150–164. Springer, Heidelberg, April 2001. [Cited on page 38.]

[140] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412, August 1999. [Cited on page 38.]

[141] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481, August 2003. [Cited on page 38.]

[142] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *CT-RSA 2005: The Cryptographer's Track at the RSA Conference*, volume 3376 of *LNCS*, pages 351–365, February 2005. [Cited on page 39.]

[143] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005: Conference on Cryptographic Hardware and Embedded Systems*, volume 3659 of *LNCS*, pages 157–171, August / September 2005. [Cited on page 39.]

[144] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In Werner Schindler and Sorin A. Huss, editors, *COSADE 2012: International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 7275 of *LNCS*, pages 69–81, May 2012. [Cited on page 39.]

[145] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications: 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers 13*, pages 64–81. Springer, 2015. [Cited on page 39.]

[146] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In Sylvain Guilley, editor, *COSADE 2017: International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 10348 of *LNCS*, pages 1–18, April 2017. [Cited on page 39.]

[147] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 321–353, April / May 2018. [Cited on page 39.]

[148] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, 2018. [Cited on page 39.]

[149] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485, April 2015. [Cited on page 39.]

[150] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: Conference on Computer and Communications Security*, pages 116–129. ACM Press, October 2016. [Cited on page 40.]

[151] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020. [Cited on page 40.]

[152] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and Higher-Order Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):173–214, 2021. [Cited on pages 40, 146, and 149.]

[153] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-Order Masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Report 2022/058, 2022. [Cited on page 40.]

[154] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked Saber. In *Security and Cryptography for Networks: 13th International Conference, SCN 2022, Amalfi (SA), Italy, September 12–14, 2022, Proceedings*, pages 93–116. Springer, 2022. [Cited on page 40.]

[155] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(2):1–26, 2021. [Cited on page 40.]

[156] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):110–145, 2023. [Cited on pages 40 and 142.]

[157] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A Configurable CRYSTALS-Kyber Hardware Implementation with Side-Channel Protection. *ACM Transactions on Embedded Computing Systems*, March 2023. [Cited on page 40.]

[158] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):414–460, 2022. [Cited on pages xxv, 40, 137, and 138.]

[159] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. A masked pure-hardware implementation of kyber cryptographic algorithm. In *International Conference on Field-Programmable Technology, (IC)FPT 2022, Hong Kong, December 5-9, 2022*, page 1. IEEE, 2022. [Cited on pages xxv, 40, and 137.]

[160] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. A Lightweight Implementation of Saber Resistant Against Side-Channel Attacks. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 224–245. Springer, 2021. [Cited on pages 40 and 137.]

[161] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 246–251. IEEE, 2004. [Cited on page 41.]

[162] Jean-Luc Danger, Sylvain Guilley, Shivam Bhasin, and Maxime Nassar. Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, pages 1–8. IEEE, 2009. [Cited on page 41.]

[163] Laurent Sauvage, Sylvain Guilley, Jean-Luc Danger, Yves Mathieu, and Maxime Nassar. Successful attack on an FPGA-based WDDL DES cryptoprocessor without place and route constraints. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 640–645. IEEE, 2009. [Cited on page 41.]

[164] Amir Moradi, Mario Kirschbaum, Thomas Eisenbarth, and Christof Paar. Masked dual-rail precharge logic encounters state-of-the-art power analysis methods. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(9):1578–1589, 2011. [Cited on page 41.]

[165] Thomas Popp, Mario Kirschbaum, Thomas Zefferer, and Stefan Mangard. Evaluation of the masked logic style MDPL on a prototype chip. In Pascal Paillier

and Ingrid Verbauwhede, editors, *CHES 2007: Conference on Cryptographic Hardware and Embedded Systems*, volume 4727 of *LNCS*, pages 81–94, September 2007. [Cited on page 41.]

[166] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021. [Cited on pages xxi, 41, 42, and 114.]

[167] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits towards automated generation of composable secure gadgets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):323–344, 2022. [Cited on pages 41 and 42.]

[168] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: Conference on Computer and Communications Security*, pages 1799–1812. ACM Press, November 2022. [Cited on pages 41, 42, and 44.]

[169] David Knichel and Amir Moradi. Composable gadgets with reused fresh masks first-order probing-secure hardware circuits with only 6 fresh masks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):114–140, 2022. [Cited on pages 42 and 44.]

[170] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, David Oswald, Wang Yao, and Zhiming Zheng. Magnifying Side-Channel Leakage of Lattice-Based Cryptosystems With Chosen Ciphertexts: The Case Study of Kyber. *IEEE Transactions on Computers*, 71(9):2163–2176, 2022. [Cited on page 43.]

[171] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):296–322, 2022. [Cited on page 43.]

[172] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. Systematic Study of Decryption and Re-encryption Leakage: The Case of Kyber. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*, volume 13211 of *Lecture Notes in Computer Science*, pages 236–256. Springer, 2022. [Cited on pages 43, 131, 148, and 149.]

[173] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):553–588, 2022. [Cited on pages 43 and 149.]

[174] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`. [Cited on page 43.]

[175] Melissa Azouaoui, Yulia Kuzovkova, Tobias Schneider, and Christine van Vredendaal. Post-quantum authenticated encryption against chosen-ciphertext side-channel attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):372–396, 2022. [Cited on page 43.]

[176] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. AGEMA: Automated Generation of Masked Hardware. `https://github.com/Chair-for-Security-Engineering/AGEMA`. [Online; accessed 31-01-2024]. [Cited on page 44.]

[177] Fabio Somenzi. Colorado University Decision Diagram. `https://github.com/ivmai/cudd`, 2012. [Online; accessed 31-01-2024]. [Cited on page 44.]

[178] Tom Van Dijk and Jaco Van De Pol. Sylvan: Multi-core decision diagrams. In *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21*, pages 677–691. Springer, 2015. [Cited on page 44.]

[179] Bernard L Welch. The generalisation of student's problems when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947. [Cited on page 44.]

[180] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015: Conference on Cryptographic Hardware and Embedded Systems*, volume 9293 of *LNCS*, pages 495–513, September 2015. [Cited on pages 44, 45, and 133.]

[181] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011. [Cited on page 44.]

[182] François-Xavier Standaert. How (not) to Use Welch's T-test in Side-Channel Security Evaluations. In *Smart Card Research and Advanced Applications: 17th International Conference, CARDIS 2018, Montpellier, France, November 12–14, 2018, Revised Selected Papers 17*, pages 65–79. Springer, 2019. [Cited on page 45.]

[183] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - verification of combined attacks automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):255–284, 2022. [Cited on pages 45 and 132.]

[184] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 787–816, December 2020. [Cited on page 45.]

[185] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER - robust verification of countermeasures against fault injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):447–473, 2021. [Cited on page 45.]

[186] Farnoud Farahmand, Viet B. Dang, Duc Tri Nguyen, and Kris Gaj. Evaluating the potential for hardware acceleration of four NTRU-based key encapsulation mechanisms using software/hardware codesign. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*, pages 23–43, 2019. [Cited on pages 50, 82, 83, and 84.]

[187] Hsin-Fu Lo, Ming-Der Shieh, and Chien-Ming Wu. Design of an efficient FFT processor for DAB system. In *ISCAS 2001: The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 4, pages 654–657 vol. 4, 2001. [Cited on page 56.]

[188] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018*, pages 77–98, 2018. [Cited on page 57.]

[189] Georg Land, Pascal Sasdrich, and Tim Güneysu. A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, volume 13173 of *Lecture Notes in Computer Science*, pages 210–230. Springer, 2021. [Cited on pages 57 and 78.]

[190] Donald E Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998. [Cited on page 57.]

[191] Daniel J. Bernstein and Tanja Lange. SUPERCOP, the System for Unified Performance Evaluation Related to Cryptographic Operations and Primitive, 2021. `https://bench.cr.yp.to/supercop.html` [Accessed 07.10.2021]. [Cited on pages 58, 106, and 149.]

[192] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323, August 1987. [Cited on page 62.]

[193] Daniel Lemire, Owen Kaser, and Nathan Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *Software: Practice and Experience*, 49(6):953–970, 2019. [Cited on page 71.]

[194] Danny Savory. SHA-512 hardware implementation in VHDL. Based on NIST FIPS 180-4. . `https://github.com/dsaves/SHA-512`. [Online; accessed 31-01-2024]. [Cited on page 75.]

[195] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013. [Cited on pages 78, 106, and 141.]

[196] Georg Land, Pascal Sasdrich, and Tim Güneysu. Dilithium-Artix7: Implementations for all round-3 parameter sets of the PQC signature scheme Dilithium, 2021. `https://github.com/Chair-for-Security-Engineering/dilithium-artix7` [Accessed 07.10.2021]. [Cited on page 78.]

[197] Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Ruggero Susella. An efficient unified architecture for polynomial multiplications in lattice-based cryptoschemes. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy-ICISSP*, pages 81–88. SciTePress, 2023. [Cited on pages 82, 83, and 84.]

[198] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. [Cited on pages 84, 95, and 98.]

[199] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022. [Cited on page 86.]

[200] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, 2019. [Cited on page 86.]

[201] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. A Unified Cryptoprocessor for Lattice-Based Signature and Key-Exchange. *IEEE Transactions on Computers*, 72(6):1568–1580, 2023. [Cited on page 86.]

[202] Peter Tummeltshammer, James C Hoe, and Markus Puschel. Time-multiplexed multiple-constant multiplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(9):1551–1563, 2007. [Cited on pages 86, 98, and 102.]

[203] Martin Kumm, Peter Zipf, Mathias Faust, and Chip-Hong Chang. Pipelined adder graph optimization for high speed multiple constant multiplication. In *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 49–52. IEEE, 2012. [Cited on pages 86, 98, and 102.]

[204] Hoang Anh Tuan, Katsuhiro Yamazaki, and Shigeru Oyanagi. Three-stage pipeline implementation for SHA2 using data forwarding. In *2008 International Conference on Field Programmable Logic and Applications*, pages 29–34. IEEE, 2008. [Cited on pages xxiv, 88, and 89.]

[205] Manoj D Rote, N Vijendran, and David Selvakumar. High performance SHA-2 core using the round pipelined technique. In *2015 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6. IEEE, 2015. [Cited on pages xxiv, 88, and 89.]

[206] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. Implementation and benchmarking of round 2 candidates in the NIST post-quantum cryptography standardization process using hardware and software/hardware co-design approaches. Cryptology ePrint Archive, Report 2020/795, 2020. [Cited on pages 95 and 96.]

[207] Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Ruggero Susella. A Flexible ASIC-oriented Design for a Full NTRU Accelerator. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 591–597, 2023. [Cited on pages 95, 96, 97, and 102.]

[208] 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. Technical report, National Institute of Standards and Technology, 2020. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`. [Cited on pages 95 and 98.]

[209] Daniel J. Bernstein. Visualizing size-security tradeoffs for lattice-based encryption. Cryptology ePrint Archive, Report 2019/655, 2019. [Cited on page 99.]

[210] Daniel J. Bernstein. A discretization attack. Cryptology ePrint Archive, Report 2020/1370, 2020. [Cited on page 99.]

[211] The Saber Team. Hardware implementations and their fair comparison. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/DUyImjnPN90/m/jZLezEPzBQAJ`, 2021. [Online; accessed 31-01-2024]. [Cited on page 102.]

[212] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking - towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15: International Conference on Applied Cryptography and Network Security*, volume 9092 of *LNCS*, pages 559–578, June 2015. [Cited on page 115.]

[213] Florian Bache and Tim Güneysu. Boolean Masking for Arithmetic Additions at Arbitrary Order in Hardware. *Applied Sciences*, 12(5):2274, 2022. [Cited on pages 115, 121, and 132.]

[214] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960. [Cited on page 121.]

[215] Felix Wegener, Lauren De Meyer, and Amir Moradi. Spin me right round rotational symmetry for FPGA-specific AES: Extended version. *Journal of Cryptology*, 33(3):1114–1155, 2020. [Cited on page 124.]

[216] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness Generation for Secure Hardware Masking - Unrolled Trivium to the Rescue. Cryptology ePrint Archive, Report 2023/1134, 2023. [Cited on pages 124 and 125.]

[217] Bohan Yang, Vladimir Rožić, Miloš Grujić, Nele Mentens, and Ingrid Verbauwhede. ES-TRNG: A high-throughput, low-area true random number generator based on edge sampling. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):267–292, 2018. [Cited on page 125.]

[218] Pierre Bayon, Lilian Bossuet, Alain Aubert, Viktor Fischer, François Poucheret, Bruno Robisson, and Philippe Maurine. Contactless electromagnetic active attack on ring oscillator based true random number generator. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 151–166. Springer, 2012. [Cited on page 125.]

[219] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC 2006: International Conference on Information Security*, volume 4176 of *LNCS*, pages 171–186, August / September 2006. [Cited on page 125.]

[220] Havard Raddum. Cryptanalytic results on trivium. `https://cosic.esat.kuleuven.be/ecrypt/stream/trivium.html`, 2006. [Online; accessed 31-01-2024]. [Cited on page 125.]

[221] Jakob Feldtkeller, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. CINI MINIS: Domain isolation for fault and combined security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: Conference on Computer and Communications Security*, pages 1023–1036. ACM Press, November 2022. [Cited on page 139.]

[222] NXP Semiconductors. EdgeLock® SE050: Plug & Trust Secure Element Family – Enhanced IoT security with high flexibility. `https://www.nxp.com/se050`, 2022. [Online; accessed 31-01-2024]. [Cited on pages 141 and 154.]

[223] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013. [Cited on page 142.]

[224] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A holistic approach towards side-channel secure fixed-weight polynomial sampling. In *PKC 2023: International Conference on Theory and Practice of Public-Key Cryptography, Part II*, LNCS, pages 94–124, May 2023. [Cited on page 142.]

[225] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. Efficiently masking polynomial inversion at arbitrary order. In *Post-Quantum Cryptography: 13th International Workshop, PQCrypto 2022, Virtual Event, September 28–30, 2022, Proceedings*, pages 309–326. Springer, 2022. [Cited on page 142.]

[226] Melissa Azouaoui, Joppe W. Bos, Björn Fay, Marc Gourjon, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Christine van Vredendaal. Surviving The FO-Calypse: Securing PQC Implementations In Practice. `https://iacr.org/submit/files/slides/2022/rwc/rwc2022/48/slides.pdf`, 2022. [Online; accessed 31-01-2024]. [Cited on page 149.]

[227] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.2)*, 2023. `https://www.sagemath.org`. [Cited on page 149.]

[228] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012. [Cited on page 154.]

[229] Stefan Kölbl, Rafael Misoczki, and Sophie Schmieg. Securing tomorrow today: Why Google now protects its internal communications from quantum threats. `https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms?hl=en`, 2022. [Online; accessed 31-01-2024]. [Cited on page 155.]