Vanessa Sereinig, BSc

Analysis and Improvements of Lattice-Based Post-Quantum Cryptosystems on Embedded Devices

Application to NTRU Prime

Master's Thesis

to achieve the university degree of
Master of Science

Master's degree programme: Computer Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Priv.-Doz. Dipl.-Ing. Dr.techn. Mario Lamberger

Institute of Applied Information Processing and Communications

Faculty of Computer Science and Biomedical Engineering

Graz, January 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Acknowledgement

I would like to thank my supervisors Christian Rechberger and Lukas Helminger for their support. Lukas, thanks for your great academic advice.

I am very grateful for the cooperation with NXP Semiconductors Gmbh & Co KG. Special thanks go to my co-supervisor Mario Lamberger, who introduced me to this fascinating topic and guided me through this thesis. I also deeply thank my other colleagues at NXP: Joppe Bos, Christine van Vredendaal, Joost Renes and Tobias Schneider. You all provided exceptional advice and interesting discussions during our countless alignment meetings throughout the last year.

I also want to thank my mother Andrea, my sister Sarah and my stepfather Franz. Mama, thanks for your continuous emotional and financial support during my studies, and already my whole life.

At last, I want to thank my boyfriend Florian with all my heart. Thanks for supporting me in every aspect of my life and for always being there for me.

Abstract

Nowadays, it is crucial to use cryptography to define secure digital processes that protect our data from unauthorised access. So-called public-key encryption can be used for data encryption, digital signatures, or key-encapsulation mechanisms. Lattice-based cryptography is a subgroup of these systems, which also promises post-quantum security as opposed to currently used public-key cryptosystems like RSA and ECC. As lattice-based cryptography is rather new, it still needs more thorough research. Besides security, the efficiency of cryptosystems plays a major role, especially on embedded systems and smart cards with limited resources.

In this thesis, we analyse performance bottlenecks using the example of the lattice-based cryptosystem NTRU Prime and try to optimise the inefficient operation of polynomial multiplication. Using the Kronecker substitution, we can convert polynomials into large integers, thus running the expensive multiplication on fast large-integer multiply-accumulate (MAC) hardware units. We implemented Kronecker substitution and three variations of it for the ring $\mathbb{Z}_q[x]/(x^p - x - 1)$ of NTRU Prime, and counted the number of multiplications and additions as a basis for comparison. This approach allows us to make runtime estimates based on the total number of operations.

Compared to the current state-of-the-art implementation, we see some improvements. For an assumed register bit-width of the MAC-unit of 256-bit, we only need 19% – 34% of the operations. For 2048-bit, we only need 6% – 31%. Our results prove that Kronecker substitution is competitive on embedded devices, and our deliberately general approach can be easily transferred to various other Lattice-based cryptosystems.

Kurzfassung

Heutzutage ist es enorm wichtig mittels Kryptografie sichere digitale Abläufe zu definieren, die unsere Daten vor fremden Augen schützen. Sogenannte Public-Key-Verschlüsselungsverfahren können zur Datenverschlüsselung, für Digitale Signaturen oder zum geschützten Schlüsselaustausch dienen. Die Lattice(,,Gitter")-basierte Kryptografie ist eine Untergruppe dieser Verfahren, die im Gegensatz zu den derzeit verwendeten Public-Key-Kryptosystemen wie RSA und ECC auch Post-Quanten Sicherheit verspricht. Da die Lattice-basierte Kryptografie ein relativ neues Forschungsgebiet ist, muss sie noch gründlicher erforscht werden. Neben der Sicherheit spielt aber auch die Effizienz der Kryptosysteme eine tragende Rolle, vorallem auf eingebetteten Systemen und Chipkarten mit begrenzten Ressourcen.
In dieser Arbeit analysieren wir Performance-Engpässe am Beispiel des Lattice-basierten Kryptosystems NTRU Prime, und versuchen die rechenintensive Operation der Polynommultiplikation zu optimieren. Mit der Kronecker Substitution können wir Polynome in große Ganzzahlen umwandeln, und die teuren Multiplikation dadurch auf schnellen Multiply-Accumulate (MAC) Hardwareeinheiten für große Ganzzahlen laufen lassen. Wir haben die Kronecker Substitution und drei Variationen davon für den Ring $\mathbb{Z}_q[x]/(x^p - x - 1)$ von NTRU Prime implementiert, und als Vergleichsbasis die Anzahl an Multiplikationen und Additionen gewählt. Diese Herangehensweise erlaubt es uns Laufzeitabschätzungen basierend auf der Anzahl dieser Operationen durchzuführen.
Im Vergleich zu optimierten Implementierungen von NTRU Prime zeigen unsere gewählten Algorithmen durchaus Verbesserungen. Bei einer angenommenen Register-Bitbreite der MAC-Einheit von 256-bit benötigen wir lediglich 19% – 34% der Zugriffe. Bei 2048-bit sind es sogar nur 6% – 31%. Unsere Ergebnisse beweisen, dass Kronecker Substitution auf eingebetteten Systemen konkurrenzfähig ist, und unser bewusst generell gehaltener Ansatz lässt sich einfach auf diverse andere Lattice-basierte Kryptosysteme übertragen.

Contents

1 Introduction

In the digital age, computers surround us in nearly every situation in life. Not only does this apply to smartphones, laptops, and our other electronic devices, but also tiny computers that are embedded in places one might not expect. These include, for example, smart gadgets, chip cards like our banking card or ID, or even health-related devices like hearing aids. With the rising demand for smart living and electronically assisted processes, consumers are more and more dependent on the various manufacturers and developers for those so-called embedded devices. However, this dependency does not only refer to usability and other functional requirements but also data security. While most users would probably not mind if someone hacked into their smart fridge and stole data about their purchasing behaviour, they would undoubtedly do if their banking account got compromised and their money was stolen. Cryptography does exactly what we need: it defines secure procedures to handle our sensitive data. To name the most important properties: if applied correctly, secure cryptographic systems (cryptosystems) maintain data confidentiality by encryption and ensure integrity (i.e. detection of manipulation) by computing checksums while also providing availability of data at any time.

Security. Proving the security of a cipher is not only challenging but impossible to do without doubts. Instead, most popular public-key cryptosystems that are believed to be secure rely on one of three hard mathematical problems. These problems are the integer factorization problem, the finite-field discrete logarithm problem, and the elliptic-curve discrete logarithm problem. There are already attacks, for example Shor's algorithm, to break schemes built on these hard problems, but with current hardware, those attacks are far from being usable in practice. So, we say that these established cryptosystems are if implemented correctly, secure because known attacks are not feasible on today's hardware.

Preparing the World for Quantum Computers. While with improving hardware, the performance of many algorithms is increased significantly, this, unfortunately, applies to attacks too. Nowadays, some groups research on the topic of so-called quantum computers, which take advantage of certain physical phenomena to achieve so far undreamed-of computational powers. Once sufficiently efficient quantum computers are built, most, if not all, public-key cryptosystems that are currently in use are basically broken. To be prepared for quantum computers, researchers in the field of post-quantum cryptography (PQC) work on developing new algorithms. These should be efficient and secure on current hardware and be designed in a way that they are still secure once the quantum computers are available. We need to design and thoroughly analyse new cryptosystems to protect our privacy for the post-quantum case.

NIST Post-Quantum Cryptography Standardization. In February 2016, the National Institute of Standards and Technology (NIST) requested cryptographers to develop and submit public-key PQC algorithms to their new standardization "competition" [Nat]. The goal is to find and specify one or more *complete and secure* schemes for digital signatures (DS), public-key encryption (PEK), and key-establishment algorithms ("key encapsulation mechanisms", KEM) that are publicly disclosed and available worldwide. The process spans over several years and consists of at least three rounds in which cryptographers are asked to participate by analysing the schemes and publishing comments. After the initial round with 69 proper submissions, the competition evolved to round two early in 2019, with 26 remaining candidates. The second round focused on performance analysis across a wide variety of systems, while the first round emphasised security and correctness analysis. On 22^{nd} of July, 2020, the 15 third-round candidates were announced. Seven of them are called finalists and will still be reviewed for the possibility of becoming standardised as a result of this third round. The remaining eight are alternate candidates that still get evaluated and are considered to have high potential to get standardised, but most probably not during this last official round.

Lattice-Based Cryptosystems. Out of the 26 second-round candidates of the aforementioned NIST PQC competition, 12 are lattice-based. Speaking of the current (third) round, 5/7 finalists and 2/8 alternate candidates are lattice-based. All these algorithms involve lattices, either in the construction itself or in the security proof. Those nine candidates all rely on one of the computationally hard lattice-based problems that are believed to be unsolvable using quantum computers. Learning with Errors (LWE) [Reg09], Module-LWE [LS15], and Ring-LWE [LPR12], as well as the "NTRU assumption", which was introduced with the NTRU cryptosystem [HPS98], are the ones the NIST candidates built upon. There are also some other hard problems, for example, the Shortest Vector Problem (SVP), the Closest Vector Problem (CVP), or the Bounded Distance Decoding (BDD). Although there are many security assumptions lattice-based schemes can be based on, many of them are quite similar to each other. Ring-LWE and Module-LWE, for example, are extensions of LWE, and the NTRU assumption is almost equivalent to the SVP in a particular class of lattices. In general, lattice-based cryptography is a relatively new field and still has to be thoroughly researched.

(Streamlined) NTRU Prime. The NTRU family is a group of several different algorithms based on the NTRU cryptosystem. It consists of NTRU NTT [LS19], NTRU Classic [HPS98] and NTRU Prime [Ber+18]. NTRU Prime is an ideal-lattice-based alternate candidate of the third round of the NIST PQC competition and falls in the category of KEMs. The developers claim that NTRU Prime has many features that avoid potential security risks in the used rings' special structures that made other ideal-lattice-based schemes prone to attacks. There are two public-key cryptosystems in the NTRU Prime family, both designed for the standard goal of IND-CCA2 security. The one we will focus on is Streamlined NTRU Prime [Ber+18].

Outline. This thesis aims to increase the performance of lattice-based cryptosystems on embedded devices by identifying a significant performance bottleneck and analysing methods to mitigate its impact. To give an overview of the mathematical constructs used throughout this thesis, we start by stating them and their used notations in Chapter 2. Afterwards, Chapter 3 introduces the reader to the backgrounds of the NTRU cryptosystem, the family of cryptosystems NTRU Prime, and our target system Streamlined NTRU Prime in Section 3.1. Here, we also identify the polynomial multiplication as the primary performance problem we want to analyse. To have an overview of different multiplication methods and their input-dependent runtimes, we describe some of the most famous ones in Section 3.2, and state which of them were already analysed for NTRU Prime in practice so far in Section 3.3. Despite its very high potential for efficient hardware implementations, the Kronecker substitution (Subsection 3.2.6) was not researched in combination with Streamlined NTRU Prime yet, which is why we decided to focus on it in this thesis. In Chapter 4, we describe a set of adaptations for Kronecker substitution, which promise even further performance improvements. In Chapter 5, we explain how we applied Kronecker substitution and its variants to our target cryptosystem. Our theoretical results are presented in Chapter 6, where we explain what methodology we chose for meaningful comparisons. We then directly compare the state-of-the-art implementation of polynomial multiplication in Streamlined NTRU Prime with our chosen techniques. We show that Kronecker substitution and its variants are indeed a powerful enhancement for polynomial multiplication on hardware with the properties we specified. To conclude this thesis, Chapter 7 summarises our findings and suggests further enhancements and future work.

2 Notation

In the following, we introduce some mathematical constructs and their respective notations that are used in this thesis. For a more in-depth explanation of the below-defined terminology, we refer to the books [HPS08; Kob94; Mao03].

Rings. A *ring* is an algebraic structure that consists of a set R for which the arithmetic operations addition $(+)$ and multiplication $(\cdot)$ are defined. A well-known example of a ring is the ring of integers, commonly denoted as $\mathbb{Z}$. Additionally, the ring of integers modulo m is denoted as $\mathbb{Z}_m$. We let the ring of integer polynomials be $\mathbb{Z}[x]$, and say that polynomials $f(x) \in \mathbb{Z}[x]$ have the form $\sum_{i=0}^{n} f_i x^i$, where $n \geq 0$ denotes the degree of the polynomial, and the f_i are its integer coefficients. We write $\mathbb{Z}_m[x]$ to indicate that the coefficients f_i are integers modulo m. Further, if we have a (polynomial) ring R we write its quotient ring by g as $R/(g)$ to be the collection of all congruence classes, where $g \in R$, and a congruence class $\bar{a}$ of some $a \in R$ is the set for all $a' \in R$ such that $a' \equiv a \pmod{g}$. Thus, $R/(g) = \{\bar{a} : a \in R\}$. To combine the above, each element of a polynomial quotient ring $\mathbb{Z}_m/(g(x))$ has a unique representative with a lower degree than $g(x)$, and whose coefficients are non-negative and smaller than m. Finally, we also mention *ideals*. An ideal is a non-empty subset I of a ring R, for which $x - y \in I, \forall x, y \in I$, and $x \cdot y \in I, \forall x \in I, \forall y \in R$, hold. The set of even numbers is an example of an ideal of $\mathbb{Z}$.

Operations in a Ring. We will denote both integer multiplication and polynomial multiplication in a ring with the operator $\cdot$, where the context follows from the description of the respective setting and involved operands. The convolution product, also written with $\cdot$, is a notation for a polynomial multiplication in a polynomial quotient ring, meaning that for two operands $a(x), b(x) \in \mathbb{Z}_m/(g(x))$, with modulus $m \in \mathbb{Z}$ and polynomial $g(x) \in \mathbb{Z}_m[x]$, their product is defined as

$c(x) = a(x) \cdot b(x) \pmod{g(x)} \pmod{m}$. Additionally, if a polynomial $f(x)$ in a ring cannot be represented by any two polynomials $d(x) \cdot e(x)$, where $d(x), e(x)$ are not units in the ring, particularly $d(x), e(x) \neq 1$, we say that $f(x)$ cannot be factored and is thus irreducible over said ring. Then, we define that a *lift* is a natural map from a polynomial quotient ring $R_m = \mathbb{Z}_m[x]/(g(x))$ to a polynomial quotient ring $R = \mathbb{Z}[x]/(g(x))$. *Lifting* an element of R_m returns its unique equivalent polynomial in R with coefficients in $\{0, 1, \cdots, m-1\}$. *Center lifting* similarly returns this same elements' unique equivalent polynomial in R, but with coefficients in $\{-(m-1)/2, \ldots, (m-1)/2\}$.

Congruency and Multiplicative Inverses. If we have two integers a, b, and a modulus m, we write $a \equiv b \pmod{m}$ to show that a and b are congruent modulo m, and similarly for polynomials. The multiplicative inverse of f in some ring is denoted as f^{-1}, and satisfies $f \cdot f^{-1} = 1$ in said ring. Similarly, this also applies to polynomials $f(x), g(x)$, where $f(x) \cdot f^{-1}(x) \equiv 1 \pmod{g(x)}$ is the multiplicative inverse of $f(x)$ modulo $g(x)$, if such a multiplicative inverse exists. While all elements of a ring have an additive inverse, they are not required to have a multiplicative one.

Fields. A ring in which every nonzero element has a multiplicative inverse is called a field $\mathbb{F}$. Additionally, if we have $\mathbb{Z}_m[x]/(p(x))$, then this is called a field if and only if $p(x)$ is irreducible in $\mathbb{Z}_m[x]$. A special kind of fields are finite fields, also called Galois fields, denoted as $\mathbb{F}_p$, where p is prime, and the field has exactly p elements. Of course, as all fields are rings, the described notations for rings can also be applied to fields.

Integer representation. Any integer A can be represented as $A = \sum_{i=0}^{n-1} a_i \beta^i$, where the integer $\beta > 0$ is its internal base, the integer $n > 0$ is its length in digits, and the $a_i \in \{0, 1, \ldots, \beta - 1\}$ are the respective digits. If we evaluate a polynomial $f(x) \in \mathbb{Z}[x]$ at a given point $x = X$, where $X \in \mathbb{Z}$ is a power of 10, we may write the resulting integer $F = f(X)$ with dividing symbols "|" to emphasise the coefficients of the evaluated polynomial. For example, we let $f(x) = 35x^2 + 14x + 30$ and evaluate at $X = 10^3$ to get $F = f(10^3) = 35|014|030$. The | aim to show the coefficients of $f(x)$ and serve as a visualisation tool for better understanding, but are not a mathematical operator.

3 Background on NTRU Prime and Multiplication Algorithms

In this chapter, we describe the cryptosystem that we focus on in this thesis: NTRU Prime. Section 3.1 summarises the NTRU family tree, Subsection 3.1.1 gives an introduction to the original NTRU public-key cryptosystem, and Subsection 3.1.2 explains how (Streamlined) NTRU Prime, which tweaks NTRU, works and what the developers' parameter recommendations are. In Subsection 3.1.3, we will also hear about polynomial multiplication and why this operation, in general, is a topic of interest for us.

Afterwards, we give an in-depth introduction to the problem of polynomial multiplication. Therefore, in Section 3.2, we provide an overview of different algorithms for fast integer (and polynomial) multiplication and show how to convert polynomial arithmetic to large integer arithmetic using Kronecker substitution.

3.1 The NTRU Family

The NTRU family consists of three cryptosystems that tweak the ring-based public-key encryption cryptosystem NTRU [HPS98], published by Hoffstein, Pipher, and Silverman in 1998. A description of the original NTRUEncrypt is given in Subsection 3.1.1.

The three main branches of the NTRU tree are:

- "NTRU Classic", which uses rings of the form $\mathbb{Z}_q[x]/(x^p - 1)$, where p is prime and q is a power of 2, and follows the original NTRU system.

- “NTRU NTT”, which uses rings of the form $\mathbb{Z}_q[x]/(x^p + 1)$, where p is a power of 2 and $q \in 1 + 2p\mathbb{Z}$ is prime. Those rings are used in typical Ring-LWE-based cryptosystems.
- “NTRU Prime”, which uses rings of the form $\mathbb{Z}_q[x]/(x^p - x - 1)$, where p and q are prime. The ring of NTRU Prime is actually a field, because, given the parameter requirements for NTRU Prime (see Subsection 3.1.2), $x^p - x - 1$ is always irreducible in $\mathbb{Z}_q[x]$. NTRU Prime is the cryptosystem we are analysing in this thesis, a detailed description of the algorithm and the designer’s recommendations is given in Subsection 3.1.2.

For a more detailed picture on the NTRU family tree, we refer to Figure 1.1. in [Ber+18].

To describe the cryptosystems we denote the two actors as $\mathcal{A}$ (*Alice*, the receiver) and $\mathcal{B}$ (*Bob*, the sender).

3.1.1 The NTRU Public-Key Cryptosystem

To use the most natural approach, we will describe the NTRU public-key cryptosystem (NTRUEncrypt) using quotient polynomial rings, but note that the underlying hard mathematical problems can also be interpreted as the shortest vector problem (SVP) [Ajt98] or closest vector problem (CVP) [Din+03] in a lattice.

The following description is taken from [HPS08].

Setting. We fix an integer $N \geq 1$ and two moduli p and q, and let

$$R = \mathbb{Z}[x]/(x^N - 1), \quad R_p = \mathbb{Z}_p[x]/(x^N - 1), \quad R_q = \mathbb{Z}_q[x]/(x^N - 1)$$

be polynomials rings. We require N and p to be prime and that $\gcd(N, q) = \gcd(p, q) = 1$. Furthermore, for any positive integers d_1, d_2, we let

$$\mathcal{T}(d_1, d_2) = \left\{ a(x) \in R : \begin{array}{l} a(x) \text{ has } d_1 \text{ coefficients equal to } 1, \\ a(x) \text{ has } d_2 \text{ coefficients equal to } -1, \\ a(x) \text{ has all other coefficients equal to } 0 \end{array} \right\},$$

where polynomials in $\mathcal{T}(d_1, d_2)$ are called *ternary polynomials.*

Public parameter creation. The public parameters are (N, p, q, d), where N, p and q have to be chosen according to the guidelines above, and d is a positive integer. To guarantee correct decryption, choose $q > (6d + 1)p$. This is done by either $\mathcal{A}$ or some trusted third party.

Key creation. $\mathcal{A}$ randomly chooses two polynomials

$$f(x) \in \mathcal{T}(d+1, d) \qquad \text{and} \qquad g(x) \in \mathcal{T}(d, d),$$

where $f(x)$ needs to be invertible in both R_q and R_p (otherwise, $\mathcal{A}$ samples a new polynomial $f(x)$). Then, $\mathcal{A}$ computes the inverses

$$F_q(x) = f(x)^{-1} \text{ in } R_q \qquad \text{and} \qquad F_p(x) = f(x)^{-1} \text{ in } R_p.$$

$\mathcal{A}$ś *public key* is $h(x) = F_q(x) \cdot g(x)$ in R_q, and their *private key* to decrypt messages is the pair $(f(x), F_p(x))$.

Encryption. $\mathcal{B}$ has a *plaintext* $m(x) \in R$ whose coefficients are between $-\frac{p}{2}$ and $\frac{p}{2}$. Then, $\mathcal{B}$ chooses a random *ephemeral key* $r(x) \in \mathcal{T}(d, d)$ and computes the ciphertext $e(x) \equiv pr(x) \cdot h(x) + m(x) \pmod q$, which is in the ring R_q.

Decryption. On receiving $e(x)$, $\mathcal{A}$ computes $a(x) \equiv f(x) \cdot e(x) \pmod q$. Then, $\mathcal{A}$ center lifts $a(x)$ to an element of R and reduces modulo p to get $b(x) \equiv F_p(x) \cdot a(x) \pmod p$. For correctly chosen parameters, $b(x)$ equals the plaintext $m(x)$ (the decryption was successful).

Security. NTRU relies on the "NTRU key recovery problem" ("NTRU assumption"), which is defined as follows:

Given $h(x)$, find ternary polynomials $f(x)$ and $g(x)$ satisfying
$f(x) \cdot h(x) \equiv g(x) \pmod q$.

Any pair of polynomials $(f(x), g(x))$ with sufficiently small coefficients and that satisfies the hidden relation

$$f(x) \cdot g(x) \equiv g(x) \pmod q$$

serves as a potential NTRU decryption key. As this problem cannot be solved by a brute-force or collision attack, and as it is (almost certainly) equivalent to solving the SVP in a certain class of lattices [HPS08], the "NTRU assumption" is believed to be a post-quantum secure hard problem.

3.1.2 Streamlined NTRU Prime

Streamlined NTRU Prime is a KEM: the sender, $\mathcal{B}$, takes a public key as input and outputs a ciphertext and session key. It uses the same field as NTRU Prime: $\mathbb{Z}_q[x]/(x^p - x - 1)$, where p and q are prime. The cryptosystem offers several implementation benefits and security advantages beyond those of the chosen ring of NTRU Prime. A nice example is the elimination of decryption failures that are a common yet annoying property of most lattice-based cryptosystems, such as the original NTRUEncrypt. The authors aim for the standard security goal of IND-CCA2 security, i.e. security against adaptive chosen-ciphertext attacks [Bel+98], at the standard 2^{128} post-quantum security level.

Parameters and setting. Due to the various possibilities the parametrisation offers, Streamlined NTRU Prime is more like a family of cryptosystems. The parameters are (p, q, t), where we require that p, q, t are positive integers, p and q are prime, $t \geq 1$, $p \geq 3t$, $q \geq 32t + 1$, and $x^p - x - 1$ is irreducible in the polynomial ring $\mathbb{Z}_q[x]$. For the setting of Streamlined NTRU Prime we denote the ring $\mathbb{Z}[x]/(x^p - x - 1)$, the ring $\mathbb{Z}_3[x]/(x^p - x - 1)$, and the field $\mathbb{Z}_q[x]/(x^p - x - 1)$ by $\mathcal{R}$, $\mathcal{R}_3$, and $\mathcal{R}_q$, respectively. An element of $\mathcal{R}$ is **small** if all of its coefficients are in $\{-1, 0, 1\}$, and a small element is ***t*-small** if exactly $2t$ of its coefficients are non-zero. Such t-small elements are also called *short polynomials*, and their Hamming Weight is obviously $2t$. The authors' case study in [Ber+18] sets the recommended parameters to $p = 761, q = 4591, t = 143$ – they introduce and recommend the cryptosystem "Streamlined NTRU Prime 4591^{761}" (sntrup761). In Table 3.1 we listed the three main parameter set proposals for Streamlined NTRU Prime in the 3^{rd}-round submission of NTRU Prime, where they added a new smaller sized set and a new larger sized set to the original recommendation.

We will continue the system's description with the parameters of Streamlined NTRU Prime 4591^{761}.

Name	**Description**	p	q	t
sntrup761	Initial 2^{nd}-round recommendation	761	4591	143
sntrup653	New smaller size	653	4621	144
sntrup857	New larger size	857	5167	161

Table 3.1: Recommended parameter sets for Streamlined NTRU Prime in the 3^{rd} round submission of NTRU Prime [Ber+20].

Key generation. $\mathcal{A}$ generates a random small element $g(x) \in \mathcal{R}$, repeating until $g(x)$ is invertible in $\mathcal{R}_3$, and a uniform random t-small element $f(x) \in \mathcal{R}$. They then compute $h(x) = g(x) \cdot 3f(x)^{-1}$ in $\mathcal{R}_q$ ($f(x)$ is invertible in $\mathcal{R}_q$ because $f(x) \neq 0$ and $t \geq 1$), and encode $h(x)$ as a string $\underline{h}$, which acts as the *public key*[1]. The secrets $f(x)$ in $\mathcal{R}$ and $g(x)^{-1}$ in $\mathcal{R}_3$ are stored for later.

Encapsulation. To generate a ciphertext, $\mathcal{B}$ decodes the public key $\underline{h}$ to obtain $h(x) \in \mathcal{R}_q$, generates a uniform random t-small element $r(x) \in \mathcal{R}$, and computes $h(x)r(x) \in \mathcal{R}_q$. They then round each coefficient of this product, viewed as an integer between $-\frac{q-1}{2}$ and $\frac{q-1}{2}$, to the nearest multiple of 3, producing[2] $c(x) \in \mathcal{R}$. $c(x)$ is then encoded as a string $\underline{c}$, and $\mathcal{B}$ hashes[3] r, obtaining a left half C ("key confirmation") and a right half K. The resulting *ciphertext* is the concatenation $C\underline{c}$, the *session key* is K.

Decapsulation. To decapsulate a ciphertext, $\mathcal{A}$ decodes $\underline{c}$ to obtain $c(x) \in \mathcal{R}$, and multiplies by $3f(x)$ in $\mathcal{R}_q$. They then view each coefficient of $3f(x)c(x)$ in $\mathcal{R}_q$ as an integer between $-(\frac{q-1}{2})$ and $(\frac{q-1}{2})$ and reduce modulo 3 to obtain a polynomial $e(x)$ in $\mathcal{R}_3$. Afterwards, $\mathcal{A}$ computes $e(x)g(x)^{-1}$ in $\mathcal{R}_3$ and lifts this product to a small polynomial $r'(x) \in \mathcal{R}$. They compute $c'(x), C'$, and K' from $r'(x)$ as for the encapsulation. If $r'(x)$ is t-small, $c'(x) = c(x)$, and $C' = C$, the

[1]Encoding of public keys as strings is a special parameter for Streamlined NTRU Prime, it allows to compress public keys for systems where q is noticeably smaller than a power of 2.

[2]In our case, for $q = 4591$, each coefficient of $c(x)$ is in $\{-\frac{q-1}{2}, \dots, -3, 0, 3, \dots, \frac{q-1}{2}\}$.

[3]The encoding of ciphertexts as string and the hash function are additional parameters for Streamlined NTRU Prime. For our case, see **encoderoundedRq** and **encodeZx**, respectively, in Figure Z.1. in [Ber+18].

output is K', the same session key as the one obtained during the encapsulation ($K' = K$). Otherwise the output is `False`.

Security of parameter choice. Just like for NTRU, the security of Streamlined NTRU Prime relies on the NTRU key recovery problem. The suggested parameters $p = 761, q = 4591, t = 143$ have an estimated pre-quantum security of 2^{248}, and an expected post-quantum security level below that, but with still comfortably enough margin above the target of 2^{128} [Ber+18].

3.1.3 Performance Bottleneck: Polynomial Multiplication

This thesis's main goal is to contribute to the efficiency of Streamlined NTRU Prime on embedded devices. Therefore we researched the main performance problem of the KEM. [Ber+18] provide a Haswell cycle count of their optimised Streamlined NTRU Prime implementation, where they counted the cycles for the encapsulation and decapsulation phase. They claim that almost 75% of the needed 157052 Haswell cycles are spent on four multiplications of polynomials modulo $x^p - x - 1$. This huge performance impact makes the polynomial multiplication a very interesting point for speed enhancement analysis in this thesis, especially for our chosen candidate cipher. After we introduce different multiplication techniques in Section 3.2, we talk about the current implementations for Streamlined NTRU Prime in Section 3.3.

3.2 Large Integer Multiplication Algorithms

This section describes some well-known and commonly used methods for fast integer and polynomials multiplications, as we want to analyse polynomial multiplication techniques for our target cipher. The multiplication algorithms we introduce are the schoolbook method (Subsection 3.2.1), Karatsuba's algorithm (Subsection 3.2.2), the Toom-Cook multiplication (Subsection 3.2.3), the Number Theoretic Transform (Subsection 3.2.4), the Schönhage-Strassen method (Subsection 3.2.5), and the Kronecker substitution (Subsection 3.2.6). A great in-depth overview of these can be found in [BZ10], which is the primary information source for this section.

The Setting. We want to multiply two integers $A = \sum_{i=0}^{m-1} a_i \beta^i$ and $B = \sum_{j=0}^{n-1} b_j \beta^j$ to get the resulting integer $C = A \cdot B = \sum_{k=0}^{m+n-1} c_k \beta^k$. The above representation of A, B, C uses their internal base $\beta > 0 \in \mathbb{Z}$, the positive integers m, n, k denote their respective amounts of digits, and $a_i, b_i, c_i \in \mathbb{Z}$ denote the values of the digits. Most of the following methods will require inputs of the same length, in which cases n denotes the length of both A and B.

All algorithms described in this section can be used for the above integer arithmetic and univariate polynomial arithmetic, as these two are analogous with some limitations. Thus, we will show the multiplication algorithms mostly with integer arithmetic notation, as this is more intuitive. When considering the inputs as polynomials, it is important to note that most fast multiplication techniques can be viewed as *evaluation-interpolation* algorithms. Here, the *evaluation* part means to evaluate the involved polynomials at certain points. *Polynomial interpolation*, on the other hand, is the process of finding the polynomial with the lowest degree that passes through all points of a given set of points. Evaluation-interpolation, the combination of these two procedures, can be used to reconstruct the result $c(x)$ of a polynomial multiplication. According to the well-known Lagrange interpolation theorem, evaluating the input polynomials $a(x)$ and $b(x)$ at $d + 1$ points, where d is the expected degree of the output polynomial, suffices to recover $c(x)$.

Comparability. To give an overview, we show and describe the different algorithms and compare their efficiency to the naive multiplication approach. To have a notion of performance, we use the *Big-O notation* $O(\cdot)$ to state the asymptotic upper runtime of the multiplication algorithms depending on the lengths of the involved integers A, B. In certain cases, we use the notation $\Theta(\cdot)$ to state both an asymptotic lower and upper bound. We shall see that the naive multiplication approach has a quadratic asymptotic runtime and is thus the most inefficient of the presented methods. All other algorithms are sub-quadratic.

3.2.1 Schoolbook Multiplication

The Schoolbook method is the common name for the naive multiplication approach as seen in Algorithm 1, which is similar to the "long multiplication" taught at school. This algorithm multiplies two integers A, B of lengths m and n, respectively, with $m <= n$. The overall runtime is $\Theta(mn)$, as $A \cdot b_j$ needs m operations, and we do that n times in the for-loop.

```
  Input: $A = \sum_{i=0}^{m-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j \in \mathbb{Z}$
1 $C \leftarrow 0$
2 for $j = 0, 1, \ldots, n-1$ do
3   $C \leftarrow C + \beta^j(A \cdot b_j)$
4 return $C := A \cdot B = \sum_{k=0}^{m+n-1} c_k\beta^k$
```

Algorithm 1: SCHOOLBOOK(A,B) [BZ10].

3.2.2 Karatsuba's Method

In 1960, Anatoly Karatsuba [KO63] discovered a fast, sub-quadratic algorithm for large integer multiplication. Until then, it was assumed that there is no way to perform an integer multiplication in sub-quadratic asymptotic time. Karatsuba's method makes use of the later defined "divide-and-conquer" principle. It reduces one multiplication of length n to three multiplications of length $\frac{n}{2}$ plus some overhead costs of $O(n)$. Both input integers A and B need to be of the same length n.

The basic idea is to represent the integers A, B as

$$A = A_1\beta^k + A_0,\ B = B_1\beta^k + B_0,$$

for some positive integer $k < n$, and requiring $A_0, B_0 < \beta^k$. The resulting product is then

$$\begin{aligned} C = A \cdot B &= (A_1\beta^k + A_0) \cdot (B_1\beta^k + B_0) \\ &= \underbrace{(A_1B_1)}_{C_2}\beta^{2k} + \underbrace{(A_1B_0 + A_0B_1)}_{C_1}\beta^k + \underbrace{(A_0B_0)}_{C_0}, \end{aligned}$$

where the four needed multiplications can be reduced to three by observing that $C_1 = (A_1 + A_0)(B_1 + B_0) - C_2 - C_0 = (A_0 - A_1)(B_1 - B_0) + C_2 + C_0$. To avoid overflows in the former rewrite of C_1, and negative coefficients in the latter rewrite, Algorithm 2 records the sign in line 6, and uses it later to sign the absolute value of C_2 in line 10 correctly.

The algorithm is outlined in Algorithm 2. It has a runtime of $\Theta(n^{\log_2 3}) \approx O(n^{1.585})$ [BZ10].

```
   Input: $A = \sum_{i=0}^{n-1} a_i\beta^i, B = \sum_{j=0}^{n-1} b_j\beta^j \in \mathbb{Z}$
1  if $n < n_0$ then
2      return Schoolbook$(A, B)$                    ▷ $n_0$ as threshold for $n$.
3  $k \leftarrow \lceil \frac{n}{2} \rceil$
4  $(A_0, B_0) \leftarrow (A \mod \beta^k, B \mod \beta^k)$
5  $(A_1, B_1) \leftarrow (\lfloor \frac{A}{\beta^k} \rfloor, \lfloor \frac{B}{\beta^k} \rfloor)$
6  $(s_A, s_B) \leftarrow (\texttt{sign}(A_0 - A_1), \texttt{sign}(B_0 - B_1))$
7  $C_0 \leftarrow$ Karatsuba$(A_0, B_0)$
8  $C_1 \leftarrow$ Karatsuba$(A_1, B_1)$
9  $C_2 \leftarrow$ Karatsuba$(|A_0 - A_1|, |B_0 - B_1|)$
10 $C \leftarrow C_0 + (C_0 + C_1 - s_A s_B C_2)\beta^k + C_1\beta^{2k}$
11 return $C := A \cdot B = \sum_{k=0}^{2n-1} c_k\beta^k$
```

Algorithm 2: Karatsuba(A, B) [BZ10].

For all needed multiplications Karatsuba$(\cdot, \cdot)$ is called recursively, except for the exit condition $n < n_0$ in line 2. Here, the threshold $n_0 \geq 2$ is used to define on what level the actual multiplications should take place, meaning at what size of the operands we swap to Schoolbook multiplication and exit the recursion. One intuitive value for n_0 would be the base of the system we operate in (i.e. β), such that digits are multiplied directly. Another way to choose n_0 is taking the hardware the algorithm is running on into account. For example, if we assume a processor that has a fast multiplier that can handle operands of size w, then setting $n_0 = w$ makes use of the available hardware best.

3.2.3 Toom-Cook Multiplication

As Karatsuba's algorithm splits the integers A and B into two parts, it is only logical to also research the possibilities for even further breaking down the inputs. In 1963, Andrei Toom [Too63] published an algorithm that involved so-called "r-way" multiplication, and Stephen Cook cleaned its description in 1966 [Coo66]. The resulting technique is called Toom-Cook Multiplication. It is, as already hinted, a generalisation of Karatsuba's algorithm. If $r = 2$, the two multiplication algorithms are the same.

Toom-Cook multiplies two integers of length n. It makes use of the evaluation-interpolation procedure: we write the two integers as $\sum_{i=0}^{r-1} a_i x^i$ and $\sum_{j=0}^{r-1} b_j x^j$ with $x = \beta^k$, and $k = \lceil \frac{n}{r} \rceil$, to get their polynomial representations $a(x)$ and $b(x)$. Their product $c(x)$ is of degree $2r - 2$, making it sufficient to evaluate at $2r - 1$ distinct points to correctly recover $c(x)$, or $c(\beta^k)$. It then takes $2r - 1$ products of input lengths of $\approx \frac{n}{r}$, instead of one product of input lengths n.

An example of Toom-Cook-3 is shown in Algorithm 3. It uses the evaluation points $0, 1, -1, 2, \infty$.

```
   Input: two integers $A, B$, with $0 \leq A, B < \beta^n$
1  if $n < 3$ then
2      return KARATSUBA$(A, B)$
3  write $A = a_0 + a_1 x + a_2 x^2,\ B = b_0 + b_1 x + b_2 x^2$, with $x = \beta^k = \beta^{\lceil \frac{n}{3} \rceil}$
4  $v_0 \leftarrow$ TOOMCOOK3$(a_0, b_0)$
5  $v_1 \leftarrow$ TOOMCOOK3$(a_0 + a_2 + a_1, b_0 + b_2 + b_1)$
6  $v_{-1} \leftarrow$ TOOMCOOK3$(a_0 + a_2 - a_1, b_0 + b_2 - b_1)$
7  $v_2 \leftarrow$ TOOMCOOK3$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
8  $v_\infty \leftarrow$ TOOMCOOK3$(a_2, b_2)$
9  $(t_1, t_2) \leftarrow (\frac{3v_0 + 2v_{-1} + v_2}{6 - 2v_\infty}, \frac{v_1 + v_{-1}}{2})$
10 $c_0 \leftarrow v_0$, $c_1 \leftarrow v_1 - t_1$, $c_2 \leftarrow t_2 - v_0 - v_\infty$, $c_3 \leftarrow t_1 - t_2$, $c_4 \leftarrow v_\infty$
11 $c(\beta^k) \leftarrow c_0 + c_1 \beta^k + c_2 \beta^{2k} + c_3 \beta^{3k} + c_4 \beta^{4k}$
12 return $c(\beta^k) := A \cdot B$
```

Algorithm 3: TOOMCOOK3(A, B) [BZ10].

Toom-Cook-3, also called Toom-3, reduces nine multiplications to five, and has an asymptotic runtime of $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.46})$ [BZ10]. The runtime generalisation for Toom-Cook-r is $\Theta(n^{\log_r(2\cdot r-1)}c(r))$, where $c(r)$ is the overhead for additions by small constants [Coo66]. The overhead $c(r)$ depends strongly on the evaluation-interpolation procedure and grows rapidly.

3.2.4 Number Theoretic Transform

The Toom-Cook r-way multiplication becomes quite complex for large r due to the quadratic increase of scalar operations ($O(r^2)$). To make the evaluation-interpolation more efficient for very large r or special points, many multiplication algorithms are based on the fast Fourier transform (FFT). The FFT applied over any finite field is called number-theoretic transform (NTT), where the used field defines the specific type. We can say that multiplications are efficiently computable with an NTT (they are *in the NTT range*) if n is large, and if the asymptotic runtime to multiply two $2n$-digit integers is roughly the same as the one to multiply two n-digit integers two times [BZ10]. This would be true for the Schönhage-Strassen algorithm, which we show in the next subsection, but not for Karatsuba or Schoolbook multiplication, for example.

Setting for the NTT: Let R be a ring, and $K \geq 2$ an integer (most commonly, K is a power of 2). We assume that there is some Ω, for which $\Omega^K = 1$ and $\sum_{i=0}^{K-1} \Omega^{ji} = 0$ for $1 \leq j < K$. Such an Ω is called a *principal K-th root of unity* in R. Performing NTT on a vector $a = [a_0, a_1, \dots, a_{K-1}]$, with $a_i \in R$, outputs the vector $\hat{a} = [\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{K-1}]$, such that its elements $\hat{a}_j = \sum_{i=0}^{K-1} \Omega^{ji} a_i$.

A recursive and in-place variant of the forward NTT can be seen in Algorithm 4. The function `bitrev(i, K)` used in line 7 returns the bit-reversal of the integer i, considered as $(\log_2 K)$-bit integer. For example, `bitrev(i, 8)` returns $0, 4, 2, 6, 1, 5, 3, 7$ for $i = 0, \dots, 7$. In addition, the backward algorithm of the NTT is given in Algorithm 5. For a proof and more detailed description of those two algorithms, we refer to [BZ10].

Algorithm 4 and Algorithm 5 both have an asymptotic runtime of $O(K \log K)$ [CCG00].

```
  Input: $K = 2^k$, for some integer $k$
  Input: vector $a = [a_0, a_1, \dots, a_{K-1}]$
  Input: $\Omega$, principal $K$-th root of unity
1 if $K = 2$ then
2     $[a_0, a_1] \leftarrow [a_0 + a_1, a_0 - a_1]$
3 else
4     $[a_0, a_2, \dots, a_{K-2}] \leftarrow$ FordwardFFT($[a_0, a_2, \dots, a_{K-2}], \Omega^2, \frac{K}{2}$)
5     $[a_1, a_3, \dots, a_{K-1}] \leftarrow$ FordwardFFT($[a_1, a_3, \dots, a_{K-1}], \Omega^2, \frac{K}{2}$)
6     for $i = 0, 1, \dots, \frac{K}{2} - 1$ do
7         $[a_{2i}, a_{2i+1}] \leftarrow [a_{2i} + \Omega^{\mathrm{bitrev}(i, \frac{K}{2})} a_{2i+1}, a_{2i} - \Omega^{\mathrm{bitrev}(i, \frac{K}{2})} a_{2i+1}]$
8 return in-place transformed vector $\hat{a}$, bit-reversed
```

Algorithm 4: FordwardFFT(a, Ω, K) [BZ10].

```
  Input: $K = 2^k$, for some integer $k$
  Input: vector $a$ of length $K$, $a = [a_0, a_{K/2}, \dots, a_{K-1}]$ (bit-reversed)
  Input: $\Omega$, principal $K$-th root of unity
1 if $K = 2$ then
2     $[a_0, a_1] \leftarrow [a_0 + a_1, a_0 - a_1]$
3 else
4     $[a_0, \dots, a_{\frac{K}{2}-1}] \leftarrow$ BackwardFFT($[a_0, \dots, a_{\frac{K}{2}-1}], \Omega^2, \frac{K}{2}$)
5     $[a_{\frac{K}{2}}, \dots, a_{K-1}] \leftarrow$ BackwardFFT($[a_{\frac{K}{2}}, \dots, a_{K-1}], \Omega^2, \frac{K}{2}$)
6     for $i = 0, 1, \dots, \frac{K}{2} - 1$ do
7         $[a_i, a_{\frac{K}{2}+i}] \leftarrow [a_i + \Omega^{-i} a_{\frac{K}{2}+i}, a_i - \Omega^{-i} a_{\frac{K}{2}+i}]$      ▷ $\Omega^{-i} = \Omega^{K-i}$
8 return in-place transformed vector $\tilde{a}$, normal order
```

Algorithm 5: BackwardFFT(a, Ω, K) [BZ10].

3.2.5 Schönhage-Strassen Algorithm

To show a very famous implementation of the NTT, we chose to give an overview of the Schönhage-Strassen algorithm. It was discovered by Arnold Schönhage and Volker Strassen in 1971 [SS71], and uses the NTT to multiply two large n-bit integers. As already mentioned before, the used ring defines the variant of the NTT. Schönhage-Strassen works in the ring $\mathbb{Z}_q$, where $q = (2^n + 1)$, meaning we want to multiply two n-bit integers modulo $2^n + 1$.

The Schönhage-Strassen algorithm, which calls the NTT-specific procedures FordwardFFT(Algorithm 4) and BackwardFFT(Algorithm 5), can be seen in Algorithm 6.

```
   Input: two $n$-bit integers $A, B$, with $0 \leq A, B < 2^n + 1$
   Input: $K = 2^k$, s.t. $K$ divides $n$ and $n = MK$
1  decompose $A = \sum_{i=0}^{K-1} a_i 2^{iM}$ with $0 \leq a_i < 2^M$ for $i < K - 1$, and
    $0 \leq a_{K-1} \leq 2^M$; similarly for $B$
2  choose $n' \geq \frac{2n}{K} + k$, with $n'$ multiple of $K$
3  $\theta \leftarrow 2^{\frac{n'}{k}}, \Omega \leftarrow \theta^2$
4  for $i = 0, 1, \ldots, K - 1$ do
5      $(a_i, b_i) \leftarrow (\theta^i a_i, \theta^i b_i) \mod (2^{n'} + 1)$
6  $a \leftarrow$ FordwardFFT$(a, \Omega, K)$, $b \leftarrow$ FordwardFFT$(b, \Omega, K)$
7  for $i = 0, 1, \ldots, K - 1$ do
8      $c_i \leftarrow a_i b_i \mod (2^{n'} + 1)$  ▷ large $n$: call NTTMulMod recursively
9  $c \leftarrow$ BackwardFFT$(c, \Omega, K)$
10 for $i = 0, 1, \ldots, K - 1$ do
11     $c_i \leftarrow \frac{c_i}{(K\theta^i)} \mod (2^{n'} + 1)$
12     if $c_i \geq (i + 1) 2^{2M}$ then
13         $c_i \leftarrow c_i - (2^{n'} + 1)$
14 $C = \sum_{i=0}^{K-1} c_i 2^{iM}$
15 return $C = A \cdot B \mod (2^n + 1)$
```

Algorithm 6: NTTMulMod(A, B, K) [BZ10].

It has a runtime complexity of $O(n \log n \log \log n)$ for a suitable n (meaning it is in the NTT range, see Subsection 3.2.4) and a corresponding NTT length $K = 2^k$ $(K \approx \sqrt{n})$ [SS71].

Schönhage-Strassen was the asymptotically fastest multiplication algorithm for large integers until 2007 when Fürer [Für07] published his new method. Fürer's algorithm has a lower asymptotic upper bound, but it is not used in practice as it only achieves advantage for astronomically large input values.

3.2.6 Kronecker Substitution

All multiplication techniques described so far can be used for polynomial arithmetic as well, as it is identical to integer arithmetic (except for carry propagation in the latter). This fundamental idea reduces polynomials to large integers using the so-called Kronecker substitution method [Kro82; Har09]. While Kronecker [Kro82] already suggested this basic idea in 1882 to reduce problems concerning multivariate polynomials to univariate polynomials, we will focus on the variant of reducing multiplications in $\mathbb{Z}[x]$ to $\mathbb{Z}$ [Har09], or $\mathbb{Z}_q[x]$ to $\mathbb{Z}$. In both cases, the necessary property is that the polynomials' coefficients are bounded by some integer q.

Our goal is to calculate

$$c(x) = a(x) \cdot b(x),$$

for polynomials $a(x), b(x), c(x) \in \mathbb{Z}[x]$ having coefficients with internal base β, by reducing this problem to

$$C = A \cdot B,$$

for $A, B, C \in \mathbb{Z}$. With Kronecker substitution the coefficients of $a(x)$ and $b(x)$ are *packed* into a large integer each by evaluating the polynomials at some carefully chosen point $x = X$. The correct choice of X is fundamental as it ensures the termination of any possible carry chain in the multiplication. Assuming both polynomials have degree d we choose $X = \beta^k > dq^2$ to be a power of the base β, because the product's coefficients are bounded by dq^2 [Har09; BZ10]. According to this lower bound we compute $X = \beta^k = \beta^{1+\lfloor \log_\beta(dq^2) \rfloor}$. To get the resulting polynomial $c(x)$ from C, its coefficients can be easily retrieved (*unpacked*) by reading blocks of length k in C.

Example in base-10. If we let $\beta = 10$, $a(x) = 871x^3 + 999x^2 + 140x + 560$ and $b(x) = 787x^3 + 960x^2 + 543x + 711$, $a(x), b(x) \in \mathbb{Z}$, we see that their coefficients are bounded by $q = 999$ and that their degree is $d = 3$. To correctly evaluate these polynomials we need $X = 10^{1+\lfloor \log_{10}(3 \cdot 999^2) \rfloor} = 10^7$, getting $a(X) = A = 871|0000999|0000140|0000560$ and $b(X) = B = 787|0000960|0000543|0000711$. Their integer product is $C = A \cdot B = 685477|1622373|1542173|1736858|1323909|0403620|0398160$, and we unpack C to retrieve the resulting polynomial $c(x) = 685477x^6 + 1622373x^5 + 1542173x^4 + 1736858x^3 + 1323909x^2 + 403620x + 398160$.

Advantages. When using Kronecker substitution, we can use existing highly optimised hardware and software solutions for large integer arithmetic instead of doing heavy computations in $\mathbb{Z}_q[x]$.

3.3 Polynomial Multiplication in NTRU Prime

As we now heard of many different algorithms to implement polynomial multiplication, we come back to our candidate cipher NTRU Prime.

Multiplications in NTRU Prime. There are a total of four polynomial multiplications in the encapsulation and decapsulation of NTRU Prime. Three of them are in $\mathcal{R}_q$: $h(x)r(x)$ in encapsulation, and $3f(x)c(x)$ and $h'(x)r'(x)$ in decapsulation. The last one, $e(x)g(x)^{-1}$ in decapsulation, is in $\mathcal{R}_3$.

Related work. In general, many lattice-based cryptosystems operate in the NTT domain, which requires carefully chosen polynomials and primes to be particularly efficient. Due to Streamlined NTRU Prime's "non NTT-friendly" polynomial and prime, Bernstein at al. [Ber+18] decided to scrap the NTT completely to avoid the large drawbacks of NTTs in this setting. For the second round of the NIST competition, they purely used Karatsuba's method (see Subsection 3.2.2) for polynomial multiplications[4]. Compared to efficient NTT based systems like [Alk+16], this method even allows for slightly faster implementations and smaller sizes in the setting of NTRU Prime.
[Che+20] provide an implementation of Streamlined NTRU Prime for an 8-bit AVR microcontroller, which shows the feasibility of high-security lattice-based cryptosystems for small embedded devices. Their implementation is optimised on assembly level and is timing invariant of secret values. For polynomial multiplication, they use Karatsuba's method in combination with an efficient modular reduction for the output coefficients.
Recently, [Alk+20] proposed two different methods to perform NTT-based polynomial multiplications in non NTT-friendly rings and apply these techniques to the ring of NTRU Prime. In one of their approaches they use Good's trick

[4]Using solely Karatsuba's method was a decision after analysing several different combinations of Karatsuba, Toom-Cook, schoolbook multiplication, and even Schönhage-Strassen.

[Goo51], and in the other one a mixed radix NTT where they focus on less memory consumption. Their implementation is implemented on the ARM Cortex-M4 microcontroller. [Alk+20] claim that their work is faster and more memory efficient than the current state-of-the art implementation on Cortex-M4 by Yang et al.[5] which is a highly optimised assembler implementation that uses Toom-Cook polynomial multiplication. As the implementation of [Alk+20] seems to be the fastest at the time, we will focus on this publication for a results comparison in Chapter 6.

Kronecker substitution with Streamlined NTRU Prime. There are no publications for Kronecker substitution within the ring of (Streamlined) NTRU Prime. We figured that applying this multiplication algorithm to Streamlined NTRU Prime is an interesting experiment with a high potential for performance enhancements on embedded devices. Packing the polynomials into big integers sounds very promising when talking about co-processors equipped with a fast hardware accelerator for large integer multiplications and additions. Large integer arithmetic is particularly interesting if we assume the word size of the multiplications' operands to be w-bit, where w is in $\{256, 512, 1024, 2048\}$, i.e. w is rather big. Important to note is that handling integers that are larger than w is easy. If the integers we retrieve from the packing in the Kronecker substitution are of size x, we simply implement one of the introduced multiplication techniques on $\frac{x}{w}$-bit words to multiply the large integers.

Related work: Kronecker substitution in other lattice-based cryptosystems. [Alb+18] analysed the possibility to implement RLWE-based schemes [LPR12] on an existing cryptographic co-processor that was initially designed for the asymmetric cryptosystem RSA [RSA78]. Their target system is the key encapsulation mechanism CRYSTALS-Kyber [Bos+18], which was a second-round NIST candidate when this paper was published and advanced to a NIST finalist since. [Alb+18] implemented optimised packing and unpacking strategies to evaluate Kronecker substitution and some of its variants presented by Harvey [Har09] (see Chapter 4). They analysed Kronecker substitution together with

[5]`https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/FHAMYa-m2hY`

Karatsuba-based polynomial multiplication and Kronecker with negated evaluation points (see Section 4.2) with Schoolbook-based polynomial multiplication. They applied all that to work in the ring of Kyber, which has the modulus $x^p + 1$, with p prime, although Kyber initially operates in the NTT domain. [Alb+18] compared their methods with the original Kyber implementation and existing RSA implementations, and other related works. The base for comparison was a commercially available smart card with the above mentioned cryptographic co-processor. They proved that lattice-based PQ cryptography could be competitive on contactless smart cards compared to existing efficient implementations for ciphers like RSA. Their contribution shows that further analysing the possibilities of Kronecker substitution is a very promising task and that Harvey's methods are very relevant for future work in this direction.

4 Multipoint Kronecker Substitution

As seen in Subsection 3.2.6 multiplication with Kronecker substitution has a major drawback: the unwanted zero-padding. When evaluating polynomials at some point $x = X$ that ensures the correct handling of carries, one inevitably introduces many zeros in the resulting integers, which in turn leads to many digit-by-digit multiplications with zeros. In the example given in Subsection 3.2.6, about $\frac{3}{4}$ of the digit-by-digit products involve zeros. To skip these redundant operations, Harvey [Har09] presented several new algorithms without sacrificing the advantages stated in Subsection 3.2.6. His methods involve the evaluation at multiple points, instead of only one, when performing Kronecker substitution.

For this chapter, we fix two polynomials $a(x), b(x) \in \mathbb{Z}[x]$ and are interested in their product $c(x)$. We assume that $a(x)$ and $b(x)$ have the same degree n, and write $a(x) = \sum_{i=0}^{n-1} a_i x^i$, and similarly for $b(x)$. Additionally, we put $e = \lceil \log_2 n \rceil$. The length of an integer M is defined to be the bit-length, i.e. the number of bits in the binary representation, which is computed with $\textsc{bit-length}(M) = 1 + \lfloor \log_2 |M| \rfloor$. We further assume that the coefficients of a and b are non-negative, are bounded by some integer $q > 0$ and have lengths of at most δ, for some integer $\delta \geq 1$. The product's coefficients are restricted to have lengths of at most $2\delta + e$ [Har09].

In the following, we will describe Harvey's carefully chosen evaluation points and discuss their performance. We focus on the integer case, where we reduce from $\mathbb{Z}[x]$ to $\mathbb{Z}$.

Comparability. To have a notion of performance, we assume that integers may be added and subtracted in $O(n)$, where n is the length of the integer, and that we may divide by a power of 2 in $O(n)$ as well. Packing and unpacking of binary strings is assumed to be possible in linear time. More precisely, these operations need $O(kc)$ each, assuming packing is the act of constructing sums of the form $\sum_{i=0}^{k-1} f_i 2^{ic}$ out of a sequence of integers $f_0, \dots, f_{k-1}$ for $0 \leq f_i < 2^c$, and unpacking means to reconstruct the sequence f_i from the sum. To directly compare the following methods, we will state the bit-lengths of the involved multiplications and additions, as well as the additional overhead that is needed for packing and unpacking.

Used bases. All the above defined values follow the assumption that we use the binary system to explain Harvey's methods in the following, as this is the relevant system on hardware. But to provide nicely readable examples, we do them in base-10. This means that the examples use $\log_{10}(\cdot)$ instead of $\log_2(\cdot)$, and $10^\ell, 10^{i\ell}$ instead of 2^ℓ, $2^{i\ell}$.

4.1 Standard Kronecker Substitution (KS1)

Let $\ell = 2\delta + e$, so that the coefficients of $c(x)$ have lengths of at most ℓ. We evaluate at $x = 2^\ell$, obtaining $a(2^\ell)$ and $b(2^\ell)$, and multiply to get $c(2^\ell) = a(2^\ell) \cdot b(2^\ell)$. By unpacking $c(2^\ell)$ we retrieve the sequence c_i.

We skip the example at this point and refer to the one in Subsection 3.2.6.

Performance. The problem of computing $c(x) = a(x) \cdot b(x)$ is reduced to multiplying two integers of length $\ell(n-1) + \delta = (2\delta + e)(n-1) + \delta$, plus the packing/unpacking overhead of $O((2\delta + e)n)$ [Har09].

4.2 Negated Evaluation Points (KS2)

This following method only works for rings in which the multiply-by-two map is injective, i.e. it does not work over a field of characteristic two. We let $\ell = \delta + \lceil \frac{e}{2} \rceil$ and evaluate at $x = 2^\ell, -2^\ell$, yielding

$$\begin{aligned} a(2^\ell) &= \sum_{i=0}^{n-1} a_i 2^{i\ell}, \\ a(-2^\ell) &= \sum_{i=0}^{n-1} (-1)^i a_i 2^{i\ell}, \end{aligned}$$

and similarly for b. There are $\lceil \frac{e}{2} \rceil$ bits of zero-padding between adjacent coefficients in $a(2^\ell)$. In $a(-2^\ell)$, the padding alternates between zero-padding and one-padding.

We perform the multiplications to obtain

$$\begin{aligned} c(2^\ell) &= a(2^\ell) \cdot b(2^\ell) \\ &= \sum_{i=0}^{2n-2} c_i 2^{i\ell}, \\ c(-2^\ell) &= a(-2^\ell) \cdot b(-2^\ell) \\ &= \sum_{i=0}^{2n-2} (-1)^{2i} c_i 2^{i\ell} = \sum_{i=0}^{2n-2} c_i 2^{i\ell}. \end{aligned}$$

Before we continue, we split c into even and odd parts $c^{(0)}$ and $c^{(1)}$,

$$c^{(0)} = \sum_{i=0}^{n-1} c_{2i} x^i, \quad c^{(1)} = \sum_{i=0}^{n-2} c_{2i+1} x^i,$$

from which we find that $c(\pm 2^\ell) = c^{(0)}(2^{2\ell}) \pm 2^\ell c^{(1)}(2^{2\ell})$. By inverting the system, this can be rewritten to

$$c^{(0)}(2^{2\ell}) = \frac{c(2^\ell) + c(-2^\ell)}{2}, \tag{4.1}$$

$$c^{(1)}(2^{2\ell}) = \frac{c(2^\ell) - c(-2^\ell)}{2 \cdot 2^\ell}. \tag{4.2}$$

To reconstruct the sequence c_i we now simply read off the even and odd coefficients of c from $c^{(0)}(2^{2\ell})$ and $c^{(1)}(2^{2\ell})$, respectively.

Example (base 10). We continue with the example polynomials $a(x) = 871x^3 + 999x^2 + 140x + 560$ and $b(x) = 787x^3 + 960x^2 + 543x + 711$, whose length $n = 3$ and coefficients are bounded by $q = 999$. We compute $\delta = 1 + \lfloor \log_{10}(|999|) \rfloor = 3$ and $e = \lceil \log_{10}(3) \rceil = 1$, and it follows that the maximum coefficient length of the product $c(x)$ is $2\delta + e = 7$. This leads to $\ell = \lceil \frac{(2\delta+e)}{2} \rceil = \lceil \frac{7}{2} \rceil = 4$, meaning we evaluate at $x = 10^4$ and $x = -10^4$. Thus, we have

$$\begin{aligned}
a(10^4) &= 871|0999|0140|0560, \\
a(-10^4) &= -870|9001|0139|9440, \\
b(10^4) &= 787|0960|0543|0711, \\
b(-10^4) &= -786|9040|0542|9289.
\end{aligned}$$

The pointwise products then are

$$\begin{aligned}
c(10^4) &= a(10^4) \cdot b(10^4) = 6856392527234669903949365981600, \\
c(-10^4) &= a(-10^4) \cdot b(-10^4) = 6853147781199932743868641981600,
\end{aligned}$$

which we further pipe into the computations

$$\begin{aligned}
c^{(0)}(10^8) &= \frac{c(10^4) + c(-10^4)}{2} = 685477|01542173|01323909|00398160, \\
10^4 c^{(1)}(10^8) &= \frac{c(10^4) - c(-10^4)}{2} = 1622373|01736858|0403620|0000,
\end{aligned}$$

from which we can read off the even- and odd-index coefficients of $c(x)$.

Performance. The problem of computing $c(x) = a(x) \cdot b(x)$ is reduced to two multiplications of integers of length $\ell(n-1) + \delta = (\delta + \lceil \frac{e}{2} \rceil)(n-1) + \delta$, plus the packing/unpacking overhead of $O((2\delta + e)n)$ [Har09].

4.3 Reciprocal Evaluation Points (KS3)

We let $\ell = \lceil \frac{2\delta+e}{2} \rceil = \delta + \lceil \frac{e}{2} \rceil$ and evaluate both at $x = 2^\ell$ and $x = 2^{-\ell}$. This yields

$$\begin{aligned} a(2^\ell) &= \sum_{i=0}^{n-1} a_i 2^{i\ell}, \\ 2^{\ell(n-1)} a(2^{-\ell}) &= \sum_{i=0}^{n-1} a_i 2^{(n-1-i)\ell} = \sum_{i=0}^{n-1} a_{n-1-i} 2^{i\ell}, \end{aligned}$$

where $2^{\ell(n-1)}$ acts as a normalising factor to push the result back to $\mathbb{Z}$. We do similarly for b. Note that in those sums there are only $\lceil \frac{e}{2} \rceil$ bits of zero-padding between the adjacent coefficients.

Then, we compute the integer products

$$c(2^\ell) = a(2^\ell) \cdot b(2^\ell) = \sum_{i=0}^{2n-2} c_i 2^{i\ell}, \tag{4.3}$$

$$2^{2\ell(n-1)} c(2^{-\ell}) = 2^{\ell(n-1)} a(2^{-\ell}) \cdot 2^{\ell(n-1)} b(2^{-\ell}) = \sum_{i=0}^{2n-2} c_{2n-2-i} 2^{i\ell}. \tag{4.4}$$

The next step is to retrieve the sequence c_i back from the sums above. This introduces some special considerations regarding two important points:

1. the c_i's overlap in both sums, and
2. carry propagation needs to be handled correctly.

The c_i's are reconstructed in an iterative way with the help of a sequence of various quantities, of which some take care of carries. The main idea is to decompose the c_i into two digits as

$$c_i = \alpha_i + \beta_i 2^\ell, \tag{4.5}$$

where $0 \le i \le 2n-2$, $0 \le \alpha_i < 2^\ell$ and $0 \le \beta_i < 2^\ell - 1$. In each reconstruction iteration i, we update the various helper variables that are needed to calculate α_i and β_i, and retrieve the value c_i.

First, we note that $a(2^\ell)$ and $b(2^\ell)$ have lengths of at most $n \cdot \ell$, which means that $c(2^\ell)$ has a length of at most $2n \cdot \ell$, and similarly for $2^{2\ell(n-1)}c(2^{-\ell})$. So, we may construct the sequences $(u_i)_{i=0}^{2n-1}, (w_i)_{i=0}^{2n-1}$ by writing the above sums in base 2^ℓ:

$$c(2^\ell) = \sum_{i=0}^{2n-1} u_i 2^{i\ell}, \tag{4.6}$$

$$2^{2\ell(n-1)}c(2^{-\ell}) = \sum_{i=0}^{2n-1} w_{2n-2-i} 2^{i\ell}, \tag{4.7}$$

where $0 \le u_i, w_i < 2^\ell$.

It follows from (4.3) and (4.6) that $\alpha_0 = u_0$ and that

$$u_{i+1} + \delta_{i+1}2^\ell = \beta_i + \alpha_{i+1} + \delta_i, \qquad 0 \le i \le 2n-2, \tag{4.8}$$

where $\delta_i \in \{0, 1\}$ is the carry generated by the right-hand side of the equation, and $\delta_0 = 0$. Similarly, it follows from (4.4) and (4.7) that $\alpha_{2n-2} = w_{2n-1}$ and that

$$w_{i+1} + \epsilon_i 2^\ell = \alpha_i + \beta_{i+1} + \epsilon_{i+1}, \qquad -1 \le i \le 2n-3, \tag{4.9}$$

where $\epsilon_i \in \{0, 1\}$ is the carry generated by the right-hand side of the equation, and $\epsilon_{2n-2} = \alpha_{-1} = 0$, and $\epsilon_{-1} = 0$.

Given the u_i and w_i, this leads to the iterative reconstruction given in Algorithm 7, which also includes the recombination phase where we recover the c_i by computing (4.5) for $0 \le i \le 2n-2$. Note that ϵ_i and δ_i were each combined to a single value ϵ and δ, respectively, to get rid of the unnecessary storage of the old values from finished iterations.

Example (base 10). We continue with the running example. For two evaluation points ℓ is again 4, meaning we evaluate at $x = 10^4$ and $x = 10^{-4}$. Thus, we have

$$\begin{aligned} a(10^4) &= 871|0999|0140|0560, \\ 10^{12}a(10^{-4}) &= 560|0140|0999|0871, \\ b(10^4) &= 787|0960|0543|0711, \\ 10^{12}b(10^{-4}) &= 711|0543|0960|0787. \end{aligned}$$

```
   Input: degree $n' \in \mathbb{Z}$, bit-length $\ell' \in \mathbb{Z}$
   Input: sequences $(u_i)_{i=0}^{2n'-1}, (w_i)_{i=0}^{2n'-1}$, with $0 \leq u_i, w_i < 2^{\ell'}$
1  Initialize the sequences $(\alpha_i)_{i=-1}^{2n'-2}$, $(\beta_i)_{i=0}^{2n'-2}$ with all zeros
2  $\alpha_0 \leftarrow u_0, \alpha_{2n'-2} \leftarrow w_{2n'-1}$
3  $\delta \leftarrow 0, \epsilon \leftarrow 0$
4  // Reconstruct:
5  for $i = 0, 1, \ldots, 2n' - 3$ do
6      $\beta_i \leftarrow (w_i - \alpha_{i-1} - \epsilon) \pmod{2^{\ell'}}$        ▷ Follows from (4.9) $\pmod{2^{\ell'}}$
7      $\alpha_{i+1} \leftarrow (u_{i+1} - \beta_i - \delta) \pmod{2^{\ell'}}$   ▷ Follows from (4.8) $\pmod{2^{\ell'}}$
8      $\delta \leftarrow (\beta_i + \alpha_{i+1} + \delta - u_{j+1})/2^{\ell'}$         ▷ Follows directly from (4.8)
9      // Computation of $\epsilon$ follows from (4.9) and the bounds for $\alpha_i, \beta_i$:
10     if $\alpha_{j+1} > w_{j+2}$ then
11         $\epsilon \leftarrow 1$
12     else
13         $\epsilon \leftarrow 0$
14 $\beta_{2n'-2} \leftarrow (w_{2n'-2} - \alpha_{2n'-3} - \epsilon) \pmod{2^{\ell'}}$
15 // Recombine:
16 for $i = 0, 1, \ldots, 2n' - 2$ do
17     $c_i \leftarrow \alpha_i + \beta_i 2^{\ell'}$
18 return $(c_i)_{i=0}^{2n'-2}$, i.e. the coefficients of $c(x) \in \mathbb{Z}[x]$
```

Algorithm 7: Iterative reconstruction algorithm with subsequent recombination phase for KS3.

The pointwise products are

$$\begin{aligned} c(10^4) &= a(10^4) \cdot b(10^4) = 68|5639|2527|2346|6990|3949|3659|8160, \\ 10^{24}c(10^{-4}) &= 10^{12}a(10^{-4}) \cdot 10^{12}b(10^{-4}) \\ &= 39|8200|3752|4082|7012|2335|2441|5477, \end{aligned}$$

for which we quickly show the first iteration of the reconstruction algorithm:

We read from $c(10^4)$ that $u_0 = 8160, u_1 = 3659, \ldots, u_{2n+1} = u_5 = 68$, and from $10^{24}c(10^{-4})$ that $w_0 = 39, w_1 = 8200, \ldots, w_5 = 5477$. We also know that

$\alpha_0 = u_0 = 8160$. Then,

$$\begin{aligned}
\beta_0 &= (w_0 - \alpha_{-1} - \epsilon) \pmod{2^\ell} = (39 - 0 - 0) \pmod{10^4} = 39,\\
\alpha_1 &= (u_1 - \beta_0 - \delta) \pmod{2^\ell} = (3659 - 39 - 0) \pmod{10^4} = 3620,\\
\delta &= (\beta_0 + \alpha_1 + \delta - u_1)/10^4 = (39 + 3620 + 0 - 3659)/10^4 = 0,\\
\epsilon &= 0,
\end{aligned}$$

from which follows that $c_0 = \alpha_0 + \beta_0 \cdot 10^4 = 8160 + 39 \cdot 10^4 = 398160$.

We omit the rest of the reconstruction at this point, as one simply has to continue iteration $i = 1$ in Algorithm 7.

Performance. The problem of computing $c(x) = a(x) \cdot b(x)$ is reduced to two multiplications of integers of length $\ell(n-1) + \delta = (\delta + \lceil \frac{e}{2} \rceil)(n-1) + \delta$, plus the packing/unpacking overhead of $O((2\delta + e)n)$ [Har09].

4.4 Four Evaluation Points (KS4): Combining the Above

We let $\ell = \lceil \frac{2\delta+e}{4} \rceil$ and evaluate at $x = 2^\ell, -2^\ell, 2^{-\ell}, -2^{-\ell}$. This gives us

$$\begin{aligned}
a(2^\ell) &= \sum_{i=0}^{n-1} a_i 2^{i\ell},\\
a(-2^\ell) &= \sum_{i=0}^{n-1} (-1)^i a_i 2^{i\ell},\\
2^{\ell(n-1)} a(2^{-\ell}) &= \sum_{i=0}^{n-1} a_{n-1-i} 2^{i\ell},\\
2^{\ell(n-1)} a(-2^{-\ell}) &= \sum_{i=0}^{n-1} (-1)^i a_{n-1-i} 2^{i\ell},
\end{aligned}$$

and similarly for b.

Then, we perform the multiplications to obtain

$$\begin{aligned} c(\pm 2^{\ell}) &= a(\pm 2^{\ell}) \cdot b(\pm 2^{\ell}), \\ 2^{2\ell(n-1)}c(\pm 2^{-\ell}) &= 2^{\ell(n-1)}a(\pm 2^{\ell}) \cdot 2^{\ell(n-1)}b(\pm 2^{\ell}). \end{aligned}$$

Afterwards, we split c into even and odd parts again, like we did in Section 4.2, leading to $c^{(0)}$ and $c^{(1)}$. We find that

$$\begin{aligned} c^{(0)}(2^{2\ell}) &= \frac{c(2^{\ell}) + c(-2^{\ell})}{2}, \\ c^{(1)}(2^{2\ell}) &= \frac{c(2^{\ell}) - c(-2^{\ell})}{2 \cdot 2^{\ell}}, \\ 2^{2\ell(n-1)}c^{(0)}(2^{-2\ell}) &= \frac{c(2^{-\ell}) + c(-2^{-\ell})}{2}, \\ 2^{2\ell(n-2)}c^{(1)}(2^{-2\ell}) &= \frac{c(2^{-\ell}) - c(-2^{-\ell})}{2 \cdot 2^{\ell}}, \end{aligned}$$

where $2^{2\ell(n-2)}$ acts as a normalising factor.

To reconstruct the sequence c_i from the results above, Algorithm 7 must be called twice, with parameters $n' = (\lfloor \frac{n}{2} \rfloor + 1)$ and $\ell' = 2 \cdot \ell$. First on $c^{(0)}(2^{2\ell})$ and $2^{2\ell(n-1)}c^{(0)}(2^{-2\ell})$ to recover the coefficients of c with an even index, and again on $c^{(1)}(2^{2\ell})$ and $2^{2\ell(n-1)}c^{(1)}(2^{-2\ell})$ to recover the coefficients with an odd index.

Example (base 10). We again continue with the example from above. Here, $\ell = \lceil \frac{(2\delta + e)}{4} \rceil = 2$, so we evaluate at $x = 10^2$, $x = 10^{-2}, x = -10^2$ and $x = -10^{-2}$. Thus, we have

$$\begin{aligned} a(10^2) &= 881004560, & b(10^2) &= 796655011, \\ a(-10^2) &= -861023440, & b(-10^2) &= -777453589, \\ 10^6 a(10^{-2}) &= 561500771, & 10^6 b(10^{-2}) &= 716526787, \\ 10^6 a(-10^{-2}) &= 558699029, & 10^6 b(-10^{-2}) &= 705665213. \end{aligned}$$

The pointwise products are

$$\begin{aligned}
c(10^2) &= a(10^2) \cdot b(10^2) = 701856697437850160, \\
c(-10^2) &= a(-10^2) \cdot b(-10^2) = 669405763641126160, \\
10^{12}c(10^{-2}) &= 10^6 a(10^{-2}) \cdot 10^6 b(10^{-2}) = 402330343342652777, \\
10^{12}c(-10^{-2}) &= 10^6 a(-10^{-2}) \cdot 10^6 b(-10^{-2}) \\
&= 394254469302178177.
\end{aligned}$$

Now we compute

$$\begin{aligned}
c^{(0)}(10^4) &= \frac{c(10^2) + c(-10^2)}{2} = 68|5631|2305|3948|8160, \\
10^{12}c^{(0)}(10^{-4}) &= \frac{10^{12}c(10^{-2}) + 10^{12}c(-10^{-2})}{2} = 39|8292|4063|2241|5477,
\end{aligned}$$

to read off the even-index coefficients of $c(x)$ using Algorithm 7. To get the odd-index coefficients, we have to retrieve them similarly using

$$\begin{aligned}
10^2 c^{(1)}(10^4) &= \frac{c(10^2) - c(-10^2)}{2} = 162|2546|6898|3620|00, \\
10^{10}c^{(1)}(10^{-4}) &= \frac{10^{12}c(10^{-2}) - 10^{12}c(-10^{-2})}{2} = 40|3793|7020|2373|00.
\end{aligned}$$

Performance. The problem of computing $c(x) = a(x) \cdot b(x)$ is reduced to four multiplications of integers of length $\ell(n-1) + \delta = (\lceil \frac{2\delta+e}{4} \rceil)(n-1) + \delta$, plus the packing/unpacking overhead of $O((2\delta + e)n)$ [Har09].

This means, instead of one multiplication with big integers (see Section 4.1), we here only need *four completely independent* multiplications with integers that are *only* $\frac{1}{4}$ *the size* of the ones used for standard Kronecker substitution, plus some overhead.

4.5 Adjustments for the Signed Case

The presented algorithms KS1, KS2, KS3, and KS4 all assume unsigned coefficients of the input polynomials that are in the interval $[0, \dots, q-1]$. In the following, we will extend those algorithms to the signed case, where we assume that the inputs are in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$. The bit-length of the coefficients δ is computed accordingly.

Determining ℓ. What changes for all algorithms is the needed evaluation point. To determine the correct ℓ for the signed case we take the same formulas as for the unsigned case (where we use the new bounds for the input coefficients, of course), but add 1 to it. This additional bit serves as a sign-bit: it allows us to differentiate between positive and negative coefficients. We can interpret negative coefficients as positive by adding $2^{\ell-1}$ to them and subtracting 1 (the carry) from the next limb. Here, we define a *limb* as any part of the resulting big integer that corresponds to a coefficient in the polynomial representation.

KS1. For standard Kronecker substitution, the only thing that changes is the unpacking of the resulting big integer $c(2^\ell)$. When we read off the coefficients c_i from it, we need to determine whether c_i is positive or negative. We do that by checking if $c_i > 2^{\ell-1}$. If yes, c_i is a negative coefficient and needs to be adjusted, and we need to keep track of the carry and add it to the next limb. The full unpacking procedure for the signed case is illustrated in Algorithm 8. Other then that we do not need to make adjustments to KS1, as the signed coefficients do not impact the packing (i.e. evaluation at a certain point 2^ℓ) of the polynomials or the actual multiplication.

KS2. For the negated evaluation points, we have to adjust the way we read off the even and odd coefficients c_i from $c^{(0)}(2^{2\ell})$ (see Equation 4.1) and $c^{(1)}(2^{2\ell})$ (see Equation 4.2). As this is the same procedure as unpacking, we just need to call Algorithm 8 on both (4.1) and (4.2), with parameter $n' = n$, to get them. The rest of the steps, including the big integer additions and multiplications, can remain the same.

```
   Input: output degree $n' \in \mathbb{Z}$; bit-length $\ell \in \mathbb{Z}$
   Input: big integer $C$ to be unpacked
1  carry $\leftarrow 0$
2  for $i = 0, 1, \dots, n' - 1$ do
3      $c_i \leftarrow C \mod 2^\ell$
4      $C \leftarrow (C - c_i)/2^\ell$
5      $c_i \leftarrow c_i +$ carry
6      if $c_i > 2^{\ell-1}$ then
7          $c_i \leftarrow c_i - 2^\ell$
8          carry $\leftarrow 1$
9      else
10         carry $\leftarrow 0$
11 return $(c_i)_{i=0}^{n'-1}$
```

Algorithm 8: Unpack(ℓ, n, C). Unpacking of a large integer for the signed case.

KS3. The reciprocal evaluation points are a bit trickier, as we need to carefully adjust Algorithm 7 to the signed case. This mainly involves re-adjusting the bounds for all involved values and all operations that depend on them during the reconstruction and recombination phases. We start by noting the new bounds $-2^{\ell-1} < \alpha_i, \beta_i < 2^{\ell-1}$ in (4.5), and $-2^{\ell-1} < u_i, w_i < 2^{\ell-1}$ in (4.6) and (4.7). Reading off the u_i and w_i is the same procedure as unpacking, so we need to call Algorithm 8 on both (4.6) and (4.7), with parameter $n' = 2n$, to get the sequences $(u_i)_{i=0}^{2n-1}$ and $(w_i)_{i=0}^{2n-1}$. Equation 4.8 and Equation 4.9 both still hold, but the computations of the $\alpha_i, \beta_i, \delta$ and ϵ in Algorithm 7 need to be adjusted to consider signed α_i and β_i. The new algorithm is given in Algorithm 9. Important to note is the new computation for ϵ, which is now in $\{-1, 0, 1\}$, to handle both the upper and lower bound for the β_i in Equation 4.9. The formula for δ stays the same, but we now have $\delta \in \{0, 1, 2\}$.

KS4. As the four evaluation point method is the combination of negated and reciprocal points, the adaptations made for KS2 and KS3 need to be considered for KS4 as well.

```
**Input:** degree $n' \in \mathbb{Z}$, bit-length $\ell' \in \mathbb{Z}$
**Input:** sequences $(u_i)_{i=0}^{2n'-1}, (w_i)_{i=0}^{2n'-1}$, with $-2^{\ell'-1} < u_i, w_i < 2^{\ell'-1}$
1  Initialize the sequences $(\alpha_i)_{i=-1}^{2n'-2}$, $(\beta_i)_{i=0}^{2n'-2}$ with all zeros
2  $\alpha_0 \leftarrow u_0, \alpha_{2n'-2} \leftarrow w_{2n'-1}$
3  $\delta \leftarrow 0, \epsilon \leftarrow 0$
4  $\beta_{max} \leftarrow 2^{\ell'-1}; \alpha_{max} \leftarrow 2^{\ell'-1}$
5  $\beta_{min} \leftarrow -\beta_{max}$
6  // Reconstruct:
7  **for** $i = 0, 1, \ldots, 2n' - 3$ **do**
8      $\beta_i \leftarrow (w_i - \alpha_{i-1} - \epsilon) \pmod{2^{\ell'}}$      ▷ Follows from (4.9) $\pmod{2^{\ell'}}$
9      **if** $\beta_i > \beta_{max}$ **then**
10         $\beta_i \leftarrow \beta_i - 2^{\ell'}$      ▷ Negative $\beta_i$
11     $\alpha_{i+1} \leftarrow (u_{i+1} - \beta_i - \delta) \pmod{2^{\ell'}}$      ▷ Follows from (4.8) $\pmod{2^{\ell'}}$
12     **if** $\alpha_{i+1} > \alpha_{max}$ **then**
13         $\alpha_{i+1} \leftarrow \alpha_{i+1} - 2^{\ell'}$      ▷ Negative $\alpha_{i+1}$
14     $\delta \leftarrow (\beta_i + \alpha_{i+1} + \delta - u_{j+1})/2^{\ell'}$      ▷ Follows directly from (4.8)
15     // Computation of $\epsilon$ follows from (4.9) and the bounds for $\alpha_i, \beta_i$:
16     **if** $w_{j+2} - \alpha_{j+1} < \beta_{min}$ **then**
17         $\epsilon \leftarrow 1$
18     **else if** $w_{j+2} - \alpha_{j+1} > \beta_{max}$ **then**
19         $\epsilon \leftarrow -1$
20     **else**
21         $\epsilon \leftarrow 0$
22 $\beta_{2n'-2} \leftarrow (w_{2n'-2} - \alpha_{2n'-3} - \epsilon) \pmod{2^{\ell'}}$
23 **if** $\beta_{2n'-2} > \beta_{max}$ **then**
24     $\beta_{2n'-2} \leftarrow \beta_{2n'-2} - 2^{\ell'}$      ▷ Negative $\beta_{2n'-2}$
25 // Recombine:
26 **for** $i = 0, 1, \ldots, 2n' - 2$ **do**
27     $c_i \leftarrow \alpha_i + \beta_i 2^{\ell'}$
28 **return** $(c_i)_{i=0}^{2n'-2}$, *i.e. the coefficients of* $c(x) \in \mathbb{Z}[x]$
```

Algorithm 9: Iterative reconstruction algorithm with subsequent recombination phase for the signed case of KS3.

5 Application Strategies and Implementation Enhancements

In this chapter, we will discuss different aspects that need to be considered when implementing Kronecker substitution and its variants to the ring of NTRU Prime on hardware.
We start with a bound study of the result of a polynomial multiplication in Section 5.1. There, depending on the input polynomials, we analyse the interval the resulting coefficients lie in. At the end of this section, we also talk about the modular reductions that we need to perform after the polynomial multiplication. In Section 5.2, we discuss the hardware assumptions we make to compare our results in the end. These assumptions are necessary for defining specific implementation details. Finally, in Section 5.3, we describe different approaches for the packing and unpacking steps of the Kronecker substitution and conclude which variants are best for our specific case.

5.1 Coefficient Bounds for Polynomial Multiplication

In Subsection 3.2.6, we already talked about the importance of carefully chosen evaluation points for Kronecker substitution so that the reconstruction of coefficients is possible without errors. For our current setting, we multiply two polynomials $a(x), b(x) \in \mathcal{R}$, where $\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1)$ is an NTRU Prime ring, to get the resulting $c(x) \in \mathcal{R}$. We are interested in the lower and upper bounds for the coefficients of $c(x)$. This is essential information for calculating the smallest possible evaluation point for Kronecker substitution that still correctly breaks any carry chain. We also need it to know how much memory we need during the multiplications.

Important notes:

- When multiplying in $\mathcal{R}$, we represent the inputs $a(x)$ and $b(x)$ as elements of $\mathbb{Z}[x]$ with p coefficients. We then first calculate $a(x) \cdot b(x) \in \mathbb{Z}[x]$ and analyse the bounds of this intermediate step. Afterwards, we reduce modulo $x^p - x - 1$, which further influences the bounds (it enlarges the interval of the coefficients). We will analyse the coefficient bounds after this reduction in the ring as well.
- Reducing in $\mathcal{R}$ means that we replace every occurrence of x^p by $x + 1$, as $x^p = x + 1 \pmod{x^p - x - 1}$.
- In Streamlined NTRU Prime, the polynomial multiplications are indeed not in $\mathcal{R}$ but in $\mathcal{R}_3$ and $\mathcal{R}_q$. However, we do not want to consider the respective modulus (3 or q) in our current setting. This is because the coefficients' bounds are obvious after reducing them by 3 or q: the lower bound is 0, and the upper bound is either $3 - 1$, or $q - 1$. Actually, we simply ignore the final reduction of the coefficients because we are interested in the bounds of the c_i's before that final step. For now, q and 3 only play a role when talking about the bounds of the input polynomials' coefficients.
- This bound study is not specific for the case of Kronecker substitution, but for polynomial multiplication in the given rings in general. We analyse the bounds to optimize the choice of evaluation points for KS1.

In the following subsections, we will analyse the bounds for the resulting coefficients first after the multiplication in $\mathbb{Z}[x]$, and then again after the reduction in $\mathcal{R}$. We will do this for several different variants of the input polynomials: for unsigned and signed coefficients, and for the multiplication by a small or short polynomial.

5.1.1 Unsigned Multiplication

We start with analysing the bounds for the multiplication of two polynomials with unsigned coefficients in both $\mathbb{Z}[x]$ and $\mathcal{R}$.

Proposition 5.1.1. *Let $a(x), b(x) \in \mathbb{Z}[x]$ be polynomials with degree strictly less than p and unsigned integer coefficients $a_i, b_i \in [0, q-1]$, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{2p-2} c_i x^i \in \mathbb{Z}[x]$ has coefficients in $[0, p(q-1)^2]$.*

Proof. As we are dealing with unsigned coefficients, the lower bound must be 0. For the upper bound study we write the resulting polynomial as

$$c(x) = \left(\sum_{i=0}^{p-1} a_i x^i\right) \cdot \left(\sum_{i=0}^{p-1} b_i x^i\right) = \sum_{k=0}^{2p-2} \sum_{i,j=0,\dots,p-1}^{i+j=k} a_j b_{k-j} x^k = \sum_{k=0}^{2p-2} c_k x^k,$$

where we immediately see that the coefficients c_k are bounded by $\sum_{i,j=0,\dots,p-1}^{i+j=k} a_j b_{k-j}$. Any value $a_j b_{k-j}$ has an upper bound of $(q-1)^2$, and as we are aggregating at most p of those values, the resulting upper bound of the c_k's is $p(q-1)^2$. This gives us the interval $[0, 1, \dots, p(q-1)^2]$ for the coefficients of $c(x) \in \mathbb{Z}[x]$. $\square$

Proposition 5.1.2. *Let $a(x), b(x) \in \mathcal{R}$ be polynomials with unsigned integer coefficients $a_i, b_i \in [0, q-1]$, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{p-1} c_i x^i$ has coefficients in $[0, 2p(q-1)^2]$.*

Proof. First, we rewrite $a(x) \cdot b(x)$ as $\sum_{i=0}^{p-1} a_i \cdot (x^i b(x))$ and show that the coefficients of $x^i b(x)$ are in $[0, 2(q-1)]$ for each i, $0 \leq i < p$. We skip the obvious case for $i = 0$, but check for $i = 1$, where we have

$$xb(x) = \sum_{j=0}^{p-1} b_j x^{j+1} \equiv b_{p-1} + (b_0 + b_{p-1})x + b_1 x^2 + \cdots + b_{p-2} x^{p-1} \pmod{x^p - x - 1}.$$

The terms $b_{p-1} \cdot 1$ and $b_{p-1} \cdot x$ above are explained by the reduction modulo $x^p - x - 1$, as for $j = p - 1$ we have $b_{p-1} x^p \equiv b_{p-1}(x+1) \pmod{x^p - x - 1}$. We easily see that $xb(x)$ has coefficients in $[0, 2(q-1)]$, with the upper bound coming from the term $|(b_0 + b_{p-1})|$. Then, we use the more general form for $2 \leq i < p$ stated in the proof for Theorem 2.1 in [Ber+18], that is

$$\begin{aligned} x^i b(x) &= \sum_{j=0}^{p-1} b_j x^{j+i} \\ &\equiv b_{p-1} + (b_{p-i} + b_{p-i-1})x + \cdots + (b_{p-1} + b_0)x^i \\ &\quad + b_1 x^{i+1} + \cdots + b_{p-i-1} x^{p-1} \pmod{x^p - x - 1}, \end{aligned} \tag{5.1}$$

where the coefficients again lie in $[0, 2(q-1)]$. Now, we take a look at $\sum_{i=0}^{p-1} a_i \cdot (x^i b(x))$. As this sum adds up at most p terms of each degree, and as $a(x)$ has coefficients of at most $q-1$, we arrive at an upper bound of $2(q-1)p(q-1) = 2p(q-1)^2$ for the resulting coefficients. The lower bound stays at 0 for unsigned coefficients, meaning each coefficient of $c(x) = a(x) \cdot b(x)$ is in $[0, 2p(q-1)^2]$. $\square$

5.1.2 Signed Multiplication

The authors of Streamlined NTRU Prime [Ber+18] chose to use signed coefficients to fit them into fewer bits. This means the coefficients a_i, b_i are in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$, where q is always odd, instead of $[0, q-1]$, i.e. the coefficients are center-lifted. We continue with the bound study for the multiplication of two polynomials with signed coefficients in $\mathbb{Z}[x]$ and in $\mathcal{R}$.

Proposition 5.1.3. *Let $a(x), b(x) \in \mathbb{Z}[x]$ be polynomials with degree at most p and signed integer coefficients $a_i, b_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{2p-2} c_i x^i \in \mathbb{Z}[x]$ has coefficients in $[-p\frac{(q-1)^2}{4}, p\frac{(q-1)^2}{4}]$.*

Proof. The proof is analogous to Proposition 5.1.1, we only need to take our new bounds for the a_i's and b_i's into account, meaning we replace all $\pm(q-1)$ with $\pm\frac{q-1}{2}$. This gives us the interval $[-p\frac{(q-1)^2}{4}, p\frac{(q-1)^2}{4}]$ for the coefficients of $c(x)$. $\square$

Proposition 5.1.4. *Let $a(x), b(x) \in \mathcal{R}$ be polynomials with signed integer coefficients $a_i, b_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{p-1} c_i x^i \in \mathcal{R}$ has coefficients in $[-p\frac{(q-1)^2}{2}, p\frac{(q-1)^2}{2}]$.*

Proof. The proof is analogous to Proposition 5.1.2 when taking the new range for the a_i's and b_i's $(0 \leq i < p)$ into account. In this setting, $x^i b(x)$ has coefficients in $[-(q-1), q-1]$ for each i, $0 \leq i < p$, again coming from the sum of at most two coefficients of $b(x)$ in Equation 5.1, where those coefficients both take extreme values. If we now consider the bounds for a_i in $\sum_{i=0}^{p-1} a_i \cdot (x^i b(x))$, and that the sum has p iterations, we get the extreme values $\pm(q-1)p\frac{(q-1)}{2} = \pm p\frac{(q-1)^2}{2}$. This equals the interval $[-p\frac{(q-1)^2}{2}, p\frac{(q-1)^2}{2}]$ for the coefficients of $c(x)$. $\square$

Signed multiplication leads to coefficients (a_i, b_i, c_i) whose intervals are only half as long as in the unsigned case.

5.1.3 Multiplication with a Small Polynomial

In Streamlined NTRU Prime (or, more specific, in NTRU and all its variants), the polynomial multiplications have exactly one operand that is small, i.e. which coefficients are in $\{-1, 0, 1\}$. In the following, we will fix the second operand $b(x)$ to be small.

Proposition 5.1.5. *Let $a(x), b(x) \in \mathbb{Z}[x]$ be polynomials with degree at most p and signed integer coefficients $a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ and $b_i \in \{-1, 0, 1\}$, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{2p-2} c_i x^i \in \mathbb{Z}[x]$ has coefficients in $[-p\frac{(q-1)}{2}, p\frac{(q-1)}{2}]$.*

Proof. The proof is analogous to Proposition 5.1.3, except that we have different bounds for the b_i's. The multiplications $a_j b_{k-j}$ in the sum now yield $-\frac{(q-1)}{2} \cdot 1$ at least, and $\frac{(q-1)}{2} \cdot 1$ at most. Again, as these values are summed up at most p times, we arrive at the interval $[-p\frac{(q-1)}{2}, p\frac{(q-1)}{2}]$ for the coefficients of $c(x)$. $\square$

Proposition 5.1.6. *Let $a(x), b(x) \in \mathcal{R}$ be polynomials with signed integer coefficients $a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ and $b_i \in \{-1, 0, 1\}$ then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{p-1} c_i x^i \in \mathcal{R}$ has coefficients in $[-p(q-1), p(q-1)]$.*

Proof. The proof is analogous to Proposition 5.1.4, but with different bounds for the b_i's. When $b(x)$ is small, then the terms $x^i b(x)$ have coefficients in $[-2, 2]$ for each i, $0 \leq i < p$. This is because the extreme values of the b_i's are -1 and 1, and as we add at most two of those values. The bounds for the a_i's are still the same as in Theorem 5.1.4, and we also still sum up at most p terms, which leads to the extreme values $\pm 2p(\frac{q-1}{2}) = \pm p(q-1)$. We arrive at the interval $[-p(q-1), p(q-1)]$ for the coefficients of $c(x)$. $\square$

5.1.4 Multiplication with a Short Polynomial

Even more efficient than the multiplication with a small polynomial is the multiplication with a t-small, or short, polynomial. The operand $b(x)$ now has coefficients in $\{-1, 0, 1\}$ and a hamming weight of $2t$, meaning $b(x)$ has exactly $2t$ coefficients that are not equal to 0.

Proposition 5.1.7. *Let $a(x), b(x) \in \mathbb{Z}[x]$ be polynomials with degree at most p and signed integer coefficients $a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ and $b_i \in \{-1, 0, 1\}$, where $b(x)$ is t-small, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{2p-2} c_i x^i \in \mathbb{Z}[x]$ has coefficients in $[-t\frac{(q-1)}{2}, t\frac{(q-1)}{2}]$.*

Proof. The first two steps of the proof are analogous to Proposition 5.1.5, as the bounds for the input coefficients did not change. But as only $2t$ of the b_i's are not equal to 0, the inner sum $\sum_{i,j=0,\dots,p-1}^{i+j=k} a_j b_{k-j} x^k$ adds up $2t$ terms at most. The reason for that is that the other $p - 2t$ terms disappear due to the involved $b_{k-j} = 0$. This leads us to the interval $[-2t\frac{(q-1)}{2}, 2t\frac{(q-1)}{2}]$ for the coefficients of $c(x)$. □

Proposition 5.1.8. *Let $a(x), b(x) \in \mathcal{R}$ be polynomials with signed integer coefficients $a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ and $b_i \in \{-1, 0, 1\}$, where $b(x)$ is t-small, then the resulting polynomial $c(x) = a(x) \cdot b(x) = \sum_{i=0}^{p-1} c_i x^i \in \mathcal{R}$ has coefficients in $[-p(q-1), p(q-1)]$.*

Proof. The first steps of the proof are analogous to Proposition 5.1.6 due to the same bounds of the input coefficients. But, when $b(x)$ is t-small we sum up at most $2t$ terms that are not equal to 0, instead of p terms. Thus, we arrive at the interval $[-2t(q-1), 2t(q-1)]$ for the coefficients of $c(x)$. □

5.1.5 Correct Choice of Bit-Length ℓ

To summarize the above stated coefficient bounds, we collect them in Table 5.1. Depending on the bounds for the coefficients of the input polynomials $a(x)$ and $b(x)$, the table lists the bounds for the coefficients of the resulting polynomial $c(x)$, where $c(x)$ is either in $\mathbb{Z}[x]$ or in $\mathcal{R}$. The upper half of the table shows all cases that were discussed in the previous subsections for input coefficients bounded by some dependence of q. The lower half, however, shows those bounds for $q = 3$ to represent the multiplications in $\mathcal{R}_3$. For this case, we omitted the multiplication with a small polynomial as this would basically be the same as the signed multiplication. We also omitted the multiplication with a short polynomial, as short polynomials are not used in multiplications in $\mathcal{R}_3$.

Setting	**Bounds of a_i, b_i**	**Bounds of c_i for $c(x) \in \mathbb{Z}[x]$**	**Bounds of c_i for $c(x) \in \mathcal{R}$**
$\textsc{Unsigned}_q$	$[0, q-1]$	$[0, p(q-1)^2]$	$[0, 2p(q-1)^2]$
$\textsc{Signed}_q$	$[-\frac{q-1}{2}, \frac{q-1}{2}]$	$[-p\frac{(q-1)^2}{4}, p\frac{(q-1)^2}{4}]$	$[-p\frac{(q-1)^2}{2}, p\frac{(q-1)^2}{2}]$
$\textsc{Small}_q$	$a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ $b(x)$ small: $b_i \in \{-1, 0, 1\}$	$[-p\frac{(q-1)}{2}, p\frac{(q-1)}{2}]$	$[-p(q-1), p(q-1)]$
$\textsc{Short}_q$	$a_i \in [-\frac{q-1}{2}, \frac{q-1}{2}]$ $b(x)$ short: $b_i \in \{-1, 0, 1\}$, $2t$ of the b_i are $\neq 0$	$[-t(q-1), t(q-1)]$	$[-2t(q-1), 2t(q-1)]$
$\textsc{Unsigned}_3$	$\{0, 1, 2\}$	$[0, 4p]$	$[0, 8p]$
$\textsc{Signed}_3$	$\{-1, 0, 1\}$	$[-p, p]$	$[-2p, 2p]$

Table 5.1: Coefficient bounds of the polynomial multiplication result $c(x)$ depending on the input polynomials $a(x)$ and $b(x)$. **Upper part:** Bounds for $a(x), b(x) \in \mathbb{Z}_q[x]$. **Lower part:** Bounds for $a(x), b(x) \in \mathbb{Z}_3[x]$ ("$q = 3$").

In Section 3.3, we already mentioned the four different polynomial multiplications in the encapsulation and decapsulation in NTRU Prime. They are, in more detail:

During encapsulation:

- $h(x) \cdot r(x)$, where $h(x) \in \mathcal{R}_q$ can be represented as its center-lift in $\mathcal{R}$, meaning its coefficients are viewed in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$, and where $r(x) \in \mathcal{R}$ is a short polynomial, meaning it has a hamming weight of $2t$ and its coefficients are in $\{-1, 0, 1\}$. This follows the setting of $\textsc{Short}_q$ in Table 5.1.

During decapsulation:

- $c(x) \cdot 3f(x)$, where $c(x) \in \mathcal{R}_q$ can be represented as its center-lift in $\mathcal{R}$, and the secret $f(x) \in \mathcal{R}$ is a short polynomial. We can rewrite this multiplication to $3(c(x) \cdot f(x))$ such that we do not have to change the bounds for the coefficients of $f(x)$. We simply multiply the result's bounds by 3 to get the correct overall bounds. This follows the setting of $\textsc{Short}_q$ in Table 5.1, where we afterwards multiply each output coefficient by 3.
- $e(x) \cdot g(x)^{-1}$, where both $e(x) \in \mathcal{R}_3$ and the secret $g(x)^{-1} \in \mathcal{R}_3$ can be represented as their center-lifts in $\mathcal{R}$. This follows the setting of $\textsc{Signed}_3$ in Table 5.1.
- $h'(x) \cdot r'(x)$, where the description of the parameters is exactly the same as for the polynomial multiplication during encapsulation. This follows the setting of $\textsc{Short}_q$ in Table 5.1.

From this enumeration of the polynomial multiplications in Streamlined NTRU Prime, we see that the most important cases in Table 5.1 are $\textsc{Short}_q$ and $\textsc{Signed}_3$, as those are the ones that are used in practice.

Determining the bit-size ℓ for Kronecker substitution. By knowing the bounds $(\min_i(c_i), \max_i(c_i))$ of the coefficients of a polynomial multiplication result, we can precisely determine the correct ℓ to use for the Kronecker substitution KS1 and its variants KS2, KS3 and KS4. On the one hand, the correct choice of ℓ is to be big enough to break any carry chain in the resulting integer by eliminating possible overlaps and "reserving enough space" per coefficient. On the other hand, ℓ should be as small as possible while still achieving said goal, because evaluation at bigger bit-sizes than necessary is very costly and needs significantly more memory.
To get the ℓ for KS1, we compute the bit-length of the extreme value $max_i(c_i)$ and add 1 to it to consider the sign bit:

$$\ell_{\text{KS1}} = \textsc{bit-size}(c_{i,max}) + 1 = \lfloor \log_2(c_{i,max}) \rfloor + 2.$$

To get the corresponding ℓ's for KS2 and KS3, we simply halve the value ℓ_{KS1}, i.e.

$$\ell_{\text{KS2}} = \ell_{\text{KS3}} = \left\lceil \frac{\ell_{\text{KS1}}}{2} \right\rceil .$$

Setting	**Choice for ℓ**					
	sntrup761 & sntrup653			sntrup857		
	KS1	KS2/3	KS4	KS1	KS2/3	KS4
UNSIGNED_q	35 (64)	18 (32)	9 (16)	36 (64)	18 (32)	9 (16)
SIGNED_q	34 (64)	17 (32)	9 (16)	35 (64)	18 (32)	9 (16)
SMALL_q	23 (32)	12 (16)	6 (8)	24 (32)	12 (16)	6 (8)
Short_q	**22 (32)**	**11 (16)**	**6 (8)**	22 (32)	11 (16)	6 (8)
UNSIGNED_3	13 (16)	7 (8)	4 (4)	13 (16)	7 (8)	4 (4)
Signed_3	**12 (16)**	**6 (8)**	**4 (4)**	12 (16)	6 (8)	4 (4)

Table 5.2: Correctly chosen evaluation points 2^ℓ for the given settings and Kronecker substitution algorithms. The table shows the different ℓ's needed for every specific case. The left-hand side shows the correct choices for ℓ for the Streamlined NTRU Prime parameter sets sntrup761 and sntrup653. They share the same optimal evaluation points. The right-hand side lists the correct choices for ℓ for the bigger parameter set sntrup857. The values marked in red are the most important cases – the two settings that we use for polynomial multiplication in Streamlined NTRU Prime, in combination with our main parameter set sntrup761. The values in brackets are the respective ℓ's rounded up to the next power of 2.

Similarly, to get the correct ℓ for KS4, we halve the value for KS2 or KS3, meaning

$$\ell_{\text{KS4}} = \left\lceil \frac{\ell_{\text{KS2}}}{2} \right\rceil = \left\lceil \frac{\ell_{\text{KS3}}}{2} \right\rceil .$$

A summary of the needed ℓ's for all the described multiplication scenarios of this section can be found in Table 5.2. The table lists the correct evaluation points for all settings, where we assume that the output polynomial $c(x)$ is in $\mathcal{R}$, i.e. we assume the slightly larger bounds for the output coefficients. To better distinguish the many different scenarios, Table 5.2 refers to the introduced setting names from Table 5.1. While we show all three of the recommended parameter sets, we highlight the most important one, sntrup761, in combination with the two settings that are included in Streamlined NTRU Prime in practice. The values in brackets in Table 5.2 are the respective ℓ rounded up to the next power of 2. These rounded ℓ's are specifically interesting, as aligning the

```
Input: modulus $m \in \{3, q\}$
Input: sequence $(c_i)_{i=0}^{2p-2}$, i.e. the coefficients of a polynomial in $\mathbb{Z}[x]$
1 // Reduce from $\mathbb{Z}[x]$ to $\mathcal{R}$
2 for $i = 2p-2, 2p-3, \ldots, p$ do
3     $c_{i-p} = c_{i-p} + c_i$
4     $c_{i-p+1} = c_{i-p+1} + c_i$
5 // Reduce from $\mathcal{R}$ to $\mathcal{R}_q$ or $\mathcal{R}_3$, respectively
6 Initialize the sequence $(c'_i)_{i=0}^{p-1}$
7 for $i = 0, 1, \ldots, p-1$ do
8     $c'_i = c_i \mod m$
9 return $(c'_i)_{i=0}^{p-1}$
```

Algorithm 10: Reduction of a polynomial in $\mathbb{Z}[x]$ to a polynomial in $\mathcal{R}_q$ or $\mathcal{R}_3$, respectively. Reduces the polynomial in the ring $x^p - x - 1$ of NTRU Prime, and further reduces the polynomials' coefficients by the given modulus m, which must be either q or 3.

evaluation points with a power of 2 allows for much more efficient packing and unpacking. In Section 5.3, we will describe different strategies for the packing and unpacking procedures, and explain why we accept a bigger (i.e. power-of-2 aligned) ℓ to gain specific implementation advantages.

5.1.6 Modular Reductions in the Ring of NTRU Prime

Another crucial part when implementing polynomial multiplications is the modular reduction at the end. The bounds of Table 5.2 do not represent the final result, because we still need to reduce the output polynomial in the ring of NTRU Prime (for which we already discussed the bounds), and further reduce its coefficients by q or 3, respectively.

Algorithm 10 shows the procedure of reducing a polynomial in $\mathbb{Z}[x]$ to a polynomial in $\mathcal{R}_q$ or $\mathcal{R}_3$, respectively. The first loop reduces the polynomial in $\mathbb{Z}[x]$ in the ring $x^p - x - 1$ to get a polynomial in $\mathcal{R}$, and the second loop reduces each coefficient by $m \in \{3, q\}$ to get a polynomial in $\mathcal{R}_q$ or $\mathcal{R}_3$. While the purpose of the second loop is clear, the reasoning behind the first loop might not be that straight-forward. We refer to the proof of Theorem 5.1.2 for an explanation of the additive terms in the first loop.

This reduction to $\mathcal{R}_q$ or $\mathcal{R}_3$, depending on the context, is necessary after unpacking the large resulting integer, because the multiplication always gives us a polynomial in $\mathbb{Z}[x]$. That is the reason why we analysed the bounds for polynomial multiplications both in $\mathbb{Z}[x]$ and in $\mathcal{R}$ in the previous subsections.

Overhead for modular reductions in the ring. In total, one instance of Algorithm 10 involves $2p-2$ additions with an operand size of ℓ-bit, and p reductions of ℓ-bit values by q or 3, respectively.

Depending on the context, Algorithm 10 may be followed by a center-lift of the coefficients. A center-lift can be achieved by further processing the coefficients c_i' at the end of the reduction algorithm with the following loop:

```
1 // center-lift of the coefficients
2 for $i = 0, 1, \ldots, p-1$ do
3     $c_i' = c_i' - \frac{m-1}{2}$
```

This adds an additional overhead of p additions of at most ℓ-bit operands.

5.2 Cryptographic Co-Processors for Public-Key Cryptography

Modern embedded devices do not only consist of a single primary processor, the CPU, but they ofte have additional co-processors that are used to supplement the functions of the CPU. Co-processors are specialised hardware components dedicated to a specific family of operations, like floating-point operations, signal processing, I/O interfacing or cryptography. These dedicated functional requirements allow hardware developers to design these building blocks to be extremely efficient.

Cryptographic co-processors are of particular interest for us. They enhance the security and performance of cryptographic operations like random number generations and hash-functions. The same holds for symmetric and asymmetric cryptosystems. For public-key cryptography, dedicated hardware is mostly optimized for large-bit integer multiplication and accumulation (addition), i.e. for the operation

$$r = a \cdot b + c,$$

where a, b and c are large integers. Such co-processors are also referred to as multiply-and-accumulate hardware units, or MAC-units. These building blocks are essential for arbitrary length multiplications as needed in both the asymmetric cryptosystems RSA [RSA78] and elliptic-curve cryptography (ECC) [Mil85; Kob87], because large-bit multiplications are very time-consuming.

When dealing with Kronecker substitution, dedicated hardware for arbitrary length multiplication and addition sounds very promising. In Section 3.3, when we talked about recent related work, we already mentioned the work of [Alb+18]. They implemented variants of Kronecker substitution in combination with the cryptosystem Kyber on a specific co-processor initially designed for RSA.

For our work, as already mentioned in Section 3.3, we will not assume specific co-processors for implementing the polynomial multiplication. However, we rather assume the word-size w_{copro} of one. Compared to the word-size w_{cpu} of the native hardware, which is usually 8, 16 or 32 bits, we will work with much larger word-sizes of dedicated co-processors for asymmetric cryptography. For an interesting comparison with existing implementations, we will assume $w_{copro} \in \{256, 512, 1024, 2048\}$-bit. Although the highest of these values might not be used in practice yet, our results will show what we could achieve with Kronecker substitution and its variants if we were to use such large-bit building blocks.

Further, we note that the actual large-integer additions and multiplications are, of course, all performed on the dedicated cryptographic co-processor. However, we assume the polynomials are stored on the native hardware. When we do a packing step, we store the large integer in words of the co-processor. After the large-integer operations are finished, we load the resulting integer into CPU words again and then perform the unpacking.

5.3 Minimizing the Kronecker Substitution's Overhead

Crucial for the running time when using Kronecker substitution are the packing and unpacking steps, which we already described in Subsection 3.2.6 and Chapter 4. The efficiency of those, of course, highly depends on the combination of the underlying hardware and the actual implementation.

Assumptions and prerequisites. We assume that the inputs for the polynomial multiplications are stored by saving their coefficients as a list in memory. This means we store a polynomial by organizing its coefficients in an array of dedicated signed integer types with certain bit sizes. We determine this size by rounding the signed bit-length of the upper coefficient bound to the next power of 2, similarly as we did for choosing ℓ. In the language C, we store coefficients with the upper bound of $\frac{q-1}{2}$ in arrays of the data type `int16`, and small and short polynomials in arrays of the data type `int8`. The output coefficients are stored in `int8`, `int16` or `int32`, depending on ℓ, before we reduce the output polynomial in the correct ring to make the coefficients smaller again. To give an example, we would define the polynomial $a(x) \in \mathbb{Z}_q[x]$ like

```
int16 a[p] = {a_0, a_1, ..., a_pmin1};,
```

where the coefficients `a_0`, ..., `a_pmin1` of type `int16` were defined beforehand, and p is the degree of the polynomial. We further assume that the native hardware has a word-size of $w_{cpu} \in \{8, 16, 32\}$, meaning the types `int8` and `int16` are easily supported.

With these specifications in mind, we will discuss different implementation considerations for packing of polynomials $a(x) \in \mathbb{Z}[x]$ in Subsection 5.3.1 and unpacking of big integers $C \in \mathbb{Z}$ in Subsection 5.3.2. We will also define the most efficient approaches that we use for our application to Streamlined NTRU Prime.

5.3.1 Packing

Packing a polynomial $a(x)$ means to concatenate its coefficients into a large integer, which we achieve by evaluating the polynomial at a chosen point 2^ℓ (we thoroughly discussed the correct choices for ℓ in Subsection 5.1.5).
In the algorithms for Kronecker substitution and its variants given in Chapter 4, we illustrated the evaluation of the polynomials as computing a sum over all coefficients. However, these large integer additions are not necessary in practice.

Unsigned Case

In the unsigned case, packing is very cheap. We simply allocate the correct amount of memory for the large integer and copy the coefficients into the right places in the co-processor's memory by cheap logical OR operations. No additions are needed.

Signed Case

The signed case needs more careful handling than the unsigned case. We cannot just copy the coefficients to the right places in memory anymore, as the evaluation of negative coefficients leads to carries. More specifically, a negative coefficient a_i leads to the following overhead:

1. **Set the sign-bit to 1.** As a_i is negative, we need to set the sign-bit by adding 2^ℓ to the coefficient: $a_i = a_i + 2^\ell$. This amounts to an ℓ-bit addition.
2. **Carry generation.** A negative coefficient triggers a carry which needs to be subtracted from the next limb: $a_{i+1} = a_{i+1} - 1$. This amounts to an ℓ-bit addition.

Overhead of intuitive packing. The *worst case* would be that every single coefficient is negative, which produces an overhead of around $2p$ ℓ-bit additions per polynomial evaluation.

Another very promising approach for polynomial packing in the signed case is proposed by Bos, Renes and van Vredendaal [BRV20]. The authors suggest to make use of the properties of the rings we operate in by getting rid of the negative coefficients and viewing them all as unsigned. Their approach is to turn signed coefficients in the interval $[-\frac{q-1}{2}, \frac{q-1}{2}]$ into unsigned ones in the interval $[q - \frac{q-1}{2}, q + \frac{q-1}{2}]$ by adding q to them. Important to note is that some of the polynomials that are multiplied are secret values in NTRU Prime and thus need to be handled with care. The secrets $f(x) \in \mathcal{R}$ and $g(x)^{-1} \in \mathcal{R}_3$ are both involved in multiplications during decapsulation. To not leak any information on those secrets based on the timing of the computations, we would need to add q to each coefficient, even the positive ones. This leads to the problem of having to increase ℓ to the next power of 2 for each of the Kronecker variants. [BRV20] noticed that there is a nice solution due to the fact that Kronecker substitution is commutative with respect to subtraction of polynomials and integers. To remedy the huge performance impact that comes when having to increase ℓ, they propose to subtract q again from the coefficients *after* the Kronecker substitution, which amounts to

$$a(2^\ell) = \sum_{i=0}^{p-1} a_i 2^{\ell i} = \sum_{i=0}^{p-1} (a_i + q) 2^{\ell i} - \sum_{i=0}^{p-1} q 2^{\ell i}.$$

The above is a big integer subtraction with the fixed term $Q = \sum_{i=0}^{p-1} q 2^{\ell i}$, which can be computed and stored in advance. Adding q to all coefficients and subtracting the large integer Q again at the end, combines the advantage of not having to take care of carries while also keeping ℓ as low as possible.

Overhead of optimized packing. If we pack a single polynomial, we need to add the value q to all coefficients a_i, which amounts to p ℓ-bit additions. The big integer subtraction at the end involves operand bit-lengths of $\ell \cdot p$. For a co-processor word size of exactly $w_{copro} = \ell$, the overhead would be around the same as for the worst case of the intuitive packing approach, namely $2p$ ℓ-bit additions/subtractions. However, the bigger the co-processor's word size, the fewer calls to the hardware we need. As we assume that w_{copro} is much larger than ℓ, this approach will always be way more efficient than the intuitive packing.

We suggest to always stick to the optimized approach to avoid any leakage of the secret data. A secret is involved in 2 of 3 multiplications during decapsulation, after all.

Pre-computing packed secrets. We can even further increase the efficiency of the packing step by not only pre-computing $Q = \sum_{i=0}^{p-1} q2^{\ell i}$ but also pre-computing the large integers that we retrieve when packing our secret values. This means that, during key generation, we not only store the secrets $f(x) \in R$ and $g(x)^{-1} \in \mathcal{R}_3$, but also perform the Kronecker substitution on them to retrieve $F \in \mathbb{Z}$ and $G^{-1} \in \mathbb{Z}$. We store those secret large integers alongside the secret polynomials to spare us the packing phase of those values during decapsulation.

5.3.2 Unpacking

We have the following setting: We pack the two polynomials $a(x), b(x) \in \mathbb{Z}[x]$ by evaluating both at 2^ℓ to get $A, B \in \mathbb{Z}$, and multiply these large integers to get $C = A \cdot B$. Finally, we need to get the coefficients of the corresponding polynomial $c(x)$ by unpacking C, a procedure for which we will now discuss different strategies. Note that C is loaded into CPU words again as soon as the large-integer operations on the co-processor are finished.

While talking about the different Kronecker variants for the signed case in Section 4.5, we already gave an intuitive algorithm for unpacking in Algorithm 8. While this algorithm was given for the signed case, it can be used for the unsigned case by simply ignoring both the handling of the carry and the computation of the if-statement. Again, we can do several optimizations for this algorithm. Most importantly is that we do not need to compute line 3 and line 4 in practice, but we simply rearrange memory instead. We will now discuss this and several other enhancements of Algorithm 8 for the unsigned and signed cases.

Unsigned Case

For the unsigned case, we read ℓ-bit chunks from C to get the unsigned coefficients of $c(x)$. By aligning ℓ with powers of 2 as recommended in Subsection 5.1.5, where we saw that we need a maximum of $\ell = 32$, we make sure that an output coefficient is either a multiple of the word size, or the other way around. This leads to three possible cases:

1. $\ell < w_{cpu}$: In this case, multiple coefficients are stored in a w_{cpu}-sized chunk of memory. For example, for $\ell = 16$ and $w_{cpu} = 32$, *two* coefficients are stored in a single w_{cpu}-bit word, i.e.
$$C = \sum_{i=0} c_i' 2^{32i},$$
where $0 \leq c_i' < 2^{32}$, and $c_i' = c_i + c_{i+1} \cdot 2^{16}$. This shows that we can read the two coefficients c_i and c_{i+1} from c_i' with simple bit shifts and masking.
2. $\ell = w_{cpu}$: When the required precision ℓ matches the hardware word size, the recovering of the coefficients is especially efficient. We read the coefficients from
$$C = \sum_{i=0} c_i' 2^{\ell i} = \sum_{i=0} c_i 2^{\ell i},$$
where $0 \leq c_i', c_i < 2^{\ell}$, which means that the c_i' are already exactly the coefficients c_i we need. We can simply reinterpret the array that stores the large integer as the sequence of coefficients of $c(x)$.
3. $\ell > w_{cpu}$: If ℓ is bigger than the hardware word size, it means that the coefficients c_i span over multiple CPU words. For example, if $\ell = 32$ and $w_{cpu} = 16$, we have
$$C = \sum_{i=0} c_i' 2^{16i},$$
where $0 \leq c_i' < 2^{16}$, and $c_i = c_i' + c_{i+1}' \cdot 2^{16}$. We again only need simple bit shifts and masking to recover the c_i's.

Aligning ℓ with powers of 2 has the additional significant advantage that an output coefficient can always be represented by one of the data types `int8`, `int16` or `int32`. This means that instead of the above mentioned shifting and masking when $\ell \neq w_{cpu}$, we simply cast (reinterpret) the memory in the code, which costs nothing.

If we consider our enhancements, then unpacking for the unsigned case boils down to *computing nothing*: just reinterpret the memory where the large integer

C is stored as an array of ℓ-bit chunks to recover the c_i's, i.e.

$$C = \sum_{i=0}^{2p-2} c_i 2^{i\ell}, \tag{5.2}$$

where $0 \leq c_i < 2^\ell$. ℓ determines the unsigned integer data type (`uint8`, `uint16`, `uint32`) we need to use for storing the c_i's.

Signed Case

Our starting point for the signed unpacking is again Algorithm 8. The recovering of the unsigned c_i's can be handled exactly like illustrated in Equation 5.2 for the unsigned case, meaning we do not need line 3 and line 4 of Algorithm 8.

Signed coefficients in Algorithm 8 are determined by checking if the unsigned c_i is greater than $2^{\ell-1}$ in line 6, i.e. if the sign-bit is set. If so, we subtract 2^ℓ from c_i to get its signed representation. We also set the carry to 1 and add it to the next coefficient c_{i+1} in line 5. Bos, Renes and van Vredendaal [BRV20] noted that the process of reinterpreting an unsigned coefficient to a signed one can be simplified significantly. If we make the fair assumption that the two's complement is used to represent signed integers in memory, the subtraction in line 7 can be omitted. The if-statement in line 6 checks if the most significant bit in c_i is set in the *unsigned* representation, and if so, makes the number negative by setting the most significant bit in the *signed* representation. The same can be achieved by simply reinterpreting the unsigned integer as a signed one in memory [BRV20].
Further, the if-statement in line 6 is also used to correctly set the carry flag. To omit this non-constant time (timing leaking) operation, [BRV20] suggests to compute the carry flag with simple OR operations on two bits. We get the relevant most significant bits in line 4 by shifting the values by $\ell - 1$ to the right.

The complete optimized unpacking procedure for the signed case is illustrated in Algorithm 11. This algorithm is timing invariant and performance enhanced.

```
    **Input:** bit-length $\ell \in \mathbb{Z}$; output degree $n \in \mathbb{Z}$
    **Input:** sequence $(c_i)_{i=0}^{n}$, i.e. the unsigned coefficients
1 carry ← 0
2 **for** $i = 0, 1, \ldots, n-1$ **do**
3     limb ← $c_i$+ carry
4     carry ← (limb $\gg (\ell - 1)$)$|(c_i \gg (\ell - 1))$
5     $c_i$ ←limb
6 **return** $(c_i)_{i=0}^{n-1}$, *i.e. the signed coefficients*
```

Algorithm 11: OPTIMIZEDUNPACK$(\ell, n, (c_i)_{i=0}^{n})$. Optimized unpacking of a large integer for the signed case. Returns the in-place adapted sequence $(c_i)_{i=0}^{n}$, where the coefficients are now signed. Taken with adaptations from [BRV20].

Overhead of optimized unpacking. Taking Algorithm 11 as a reference, the optimized unpacking involves only one major operation: the carry addition in line 3. In the *worst case*, i.e. if all coefficients are negative, this amounts to $n = 2p - 2$ additions with an operand bit-size of ℓ.

Enhancements for KS3. All the enhancements for the unpacking strategies we just discussed can also be applied to the reconstruction algorithm of the signed case of KS3, which we discussed in Section 4.5.
Firstly, as already described, we get the sequences $(u_i)_{i=0}^{2L'-1}$ and $(w_i)_{i=0}^{2L'-1}$ by performing two unpacking steps. For this, we can now apply the enhanced unpacking strategies in Algorithm 11.
Secondly, in line 10, line 13, and line 24 of Algorithm 9, we check if $\beta_i > \beta_{max}$, and if so, subtract $2^{\ell'}$ from it. Similarly for α_i. These three if-statements and the corresponding subtractions by $2^{\ell'}$ can be omitted, because we can use the trick with the reinterpretation of the memory again. We just cast the unsigned integers β_i and α_i to signed integers, similarly like we described previously for the unpacking algorithm. This enhances Algorithm 9 significantly.

5.3.3 Summary of Findings

In this section, we discussed different important aspects when implementing the packing and unpacking steps of the Kronecker substitution and its variants. Our most interesting analyses are those of the signed case. We conclude that the best choice of ℓ is not necessarily the smallest possible, but the next bigger power of 2, confirming the statements in Table 5.2. With this careful selection of ℓ, we arrived at enhanced packing and unpacking methods, in which several computations could be omitted. Additionally, we gave an overview of the needed operations for the optimized packing and unpacking procedures, which help estimate the Kronecker substitution's overall performance in the next chapter.

6 Results

In this chapter, we will present our results and discuss them. We also compare them to current state-of-the-art implementations of polynomial multiplications in Streamlined NTRU Prime.

Setting. We assume an embedded system that consists of a CPU and one or more co-processors to supplement the main computer. Most important is a cryptographic co-processor, i.e. a fast large-bit multiply-accumulate hardware (MAC) unit on which we can run our large-integer arithmetic. We have no limitations on the word size w_{cpu} of the CPU; it can be anywhere in the common range of 8 to 64 bits. What matters is the word size of the MAC unit w_{copro}, which is the main basis for our results and comparisons. We will assume word sizes w_{copro} of 256, 512, 1024 and 2048 bits for our later explained methodology. Word sizes of 128 to 512 bits are fairly common in co-processors for asymmetric cryptography, but we also want to look at the performance of our implementation on much larger units.

Counting multiplications and additions. We are interested in a high-level count of x-bit integer additions and multiplications, where x-bit refers to the operand bit-length of said operations. The bit-lengths of these operations highly depend on the respective ℓ of the chosen Kronecker substitution variant.
When we talk about operation counts, we always mean the number of x-bit additions and multiplications needed for a single polynomial multiplication in NTRU Prime, not those of a full encapsulation or decapsulation phase.
To get the counts for Kronecker substitution and its variants, we counted the operations in theory and verified these results in our implementation.
To have a fair basis for comparison, we always assume the worst-case scenarios (i.e., inputs) for polynomial multiplications when evaluating our methods. When

counting the operations needed for packing and unpacking as described in Section 5.3, for example, we count the operations as if each coefficient was negative – although this is never the case in practice. The same holds for other parts where negative coefficients lead to more overhead or any other timing-variant parts of the algorithms.
Our counts for KS1, KS2, KS3 and KS4 are given in Table 6.1. The bold, red rows emphasise the most important operations, i.e. the actual big-integer additions and multiplications needed for the individual setting. All red lines, in general, show big-integer arithmetic that should be performed on the cryptographic co-processor. All other operations are small enough to be efficiently performed on the main computer.

Comparison methodology. How many additions and multiplications we need to perform in practice depends on the combination of the two scenarios described above: the word size of the co-processor and the bit-lengths of the operands. If the bit-sizes x of the operands are larger than w_{copro}, we actually need more than one operation to get our result. How many exactly depends on the method we choose, i.e. which of the techniques in Section 3.2 we use for big-integer multiplication. Again we assume a method with the worst runtime to give the fairest count: the schoolbook method. This means that for an x-bit multiplication, where $x > w_{copro}$, we assume that we actually need $\lceil \left(\frac{x}{w_{copro}}\right)^2 \rceil$ multiplications on the co-processors. For additions, we similarly use the respective schoolbook variant. Schoolbook addition has an asymptotic runtime complexity of $O(n)$, where n is the bit-length of the operands. So, for x-bit additions, where $x > w_{copro}$, we assume we need $\lceil \frac{x}{w_{copro}} \rceil$ additions on the co-processor.
We calculate the actual addition and multiplication counts for every performed big-integer operation in Table 6.1 for all $w_{copro} \in \{256, 512, 1024, 2048\}$, and sum them up. The resulting counts are given in Table 6.2. Note that this table does not include the small (≤ 64 bits) additions from Table 6.1, as we do not perform them on the co-processor, but directly on CPU (or anywhere else). Table 6.2 only includes the needed additions and multiplications that we run on the cryptographic co-processor with its respective word size w_{copro}.

Method	**Operation (Context)**	**Operand bit-size**	**Count**
KS1: $\ell = 32$	**MUL**	**24333**	**1**
	ADD (pack: add q to coefficients)	32	1522
	ADD (pack: subtract big integer Q)	24334	2
	ADD (unpack: add carry)	32	1521
	ADD (reduction)	32	1520
KS2: $\ell = 16$	**MUL**	**12173**	**2**
	ADD	**24333**	**2**
	ADD (pack: add q to coefficients)	16	3044
	ADD (pack: subtract big integer Q)	12174	4
	ADD (unpack: add carry)	32	1522
	ADD (reduction)	32	1520
KS3: $\ell = 16$	**MUL**	**12173**	**2**
	ADD (pack: add q to coefficients)	16	3044
	ADD (pack: subtract big integer Q)	12174	4
	ADD (unpack: reconstruction alg.)	16	15206
	ADD (reduction)	32	1520
KS4: $\ell = 8$	**MUL**	**6093**	**4**
	ADD	**12173**	**4**
	ADD (pack: add q to coefficients)	32	6088
	ADD (pack: subtract big integer Q)	6094	8
	ADD (unpack: reconstruction alg.)	16	15204
	ADD (reduction)	16	1520

Table 6.1: Addition and multiplication counts for the Kronecker substitution and its variants. Counts for packing and unpacking overhead are according to the discussed optimised algorithms. Each red row indicates an operation that should be computed on a cryptographic co-processor for fast large-bit integer arithmetic. The bold, red rows emphasise the main multiplications and additions for the respective Kronecker substitution variant.

Method	**Operation**	**Count**
KS1	MUL	9035
$\ell = 32$	ADD	192
KS2	MUL	4523
$\ell = 16$	ADD	383
KS3	MUL	4523
$\ell = 16$	ADD	192
KS4	MUL	2266
$\ell = 8$	ADD	383

Left: $w_{copro} = 256$ bits.

Method	**Operation**	**Count**
KS1	MUL	2259
$\ell = 32$	ADD	96
KS2	MUL	1131
$\ell = 16$	ADD	192
KS3	MUL	1131
$\ell = 16$	ADD	96
KS4	MUL	567
$\ell = 8$	ADD	192

Right: $w_{copro} = 512$ bits.

Method	**Operation**	**Count**
KS1	MUL	565
$\ell = 32$	ADD	48
KS1	MUL	283
$\ell = 16$	ADD	96
KS1	MUL	283
$\ell = 16$	ADD	48
KS1	MUL	142
$\ell = 8$	ADD	96

Left: $w_{copro} = 1024$ bits.

Method	**Operation**	**Count**
KS1	MUL	142
$\ell = 32$	ADD	24
KS1	MUL	71
$\ell = 16$	ADD	48
KS1	MUL	71
$\ell = 16$	ADD	24
KS1	MUL	36
$\ell = 8$	ADD	48

Right: $w_{copro} = 2048$ bits.

Table 6.2: Addition and multiplication counts for big-integer operations performed on a fast large-bit cryptographic co-processor, i.e. the red rows in Table 6.1. Different sub-tables show the counts for different assumed word sizes w_{copro} of the co-processor. For both large-bit addition and multiplication, we assume schoolbook methods to estimate the total counts.

.

State-of-the-art comparison. We compare the counts of our methods with the most recent optimised implementation of Streamlined NTRU Prime published by [Alk+20]. The authors state that one instance of a polynomial multiplication in their mixed-radix NTT involves around 56322 32-bit multiplications and 17820 32-bit additions[1].

Results comparison and discussion. A summary of the main findings of this thesis are given in Table 6.3. This table includes the direct comparison of all needed additions and multiplications for our implementations and the one from [Alk+20], and the comparison of all $w_{copro} \in \{256, 512, 1024, 1048\}$. For a better overview, we combine the total counts of additions and multiplications (as given in Table 6.2) and simply call them *operations*. We also give the direct difference in % compared to the reference from [Alk+20], where a change of $-x\%$ indicates that our variants need $x\%$ less total operations to perform the same polynomial multiplication. To keep the interesting overview of operations performed by the CPU and the ones performed by the MAC unit (the threshold for CPU computation is ≤ 64 bits), we keep them separate. We assume a CPU word size w_{cpu} of 64 bits. Note that the implementation of [Alk+20] involves only 32-bit operations, so we assumed they do not make use of the cryptographic co-processor at all.
We see that the operation counts for our results are much better than for the current state-of-the-art implementation, even for the smallest assumed word size of the co-processor of 256 bits. With increasing w_{copro}, the total number of needed operations decreases. However, the amount of operations with small bit-lengths is always the same, and those operations also contribute most to the total numbers. For $w_{copro} = 4096$, the small operations represent around $96 - 98\%$ of the total operations for KS1 and KS2. For KS3 and KS4, they even make up more than 99% of the total operations. Here, we keep our possible overhead improvements from Section 5.3 in mind. If we already pack the secrets and store those large integers in advance, we can save some packing overhead during decapsulation.
What does change significantly with increasing w_{copro} is the number of operations that need to be performed on the cryptographic co-processor. As expected,

[1]The authors of [Alk+20] have communicated to us the multiplication and addition counts for their mixed-radix case. This count reflects a single polynomial multiplication including the overhead for the NTT and the inverse (backward) NTT.

KS1 and KS2 give the most promising operation counts for all assumed w_{copro}'s. That KS1 is that compatible to KS2 is a rather surprising result. This is due to the huge overhead of the small operations, as already discussed. In comparison, KS3 and KS4 perform poorly, because of the complex reconstruction algorithm (Algorithm 9) that those Kronecker substitution variants involve. This reconstruction procedure increases the overhead of small operations significantly, as one can see in the resulting operation counts. Furthermore, the additional sequences that are processed in Algorithm 9 (u, w, α, β) add considerable memory needs for KS3 and KS4. On embedded systems, we should keep the code size and the consumed runtime memory as low as possible, which KS3 and KS4 do not fulfill.
Due to the listed reasons, we do not recommend to use KS3 or KS4 in practice, but to use KS2 or KS1. The best approach depends on the specific word size for the small operations, i.e. the best Kronecker substitution variant is the one where $w_{cpu} = \ell$, or slightly larger.

Omitted operations. As thoroughly described in Section 5.3, we highly optimised the procedures of packing and unpacking. Those enhanced algorithms include some operations that we consider very cheap and that we therefore did not count. This includes operations on the bit level, like bit-wise shifts and OR operations. For example, every call to Algorithm 11 has a small overhead of n OR operations and $2n$ right-shifts by $\ell - 1$. Furthermore, we have some additional overhead if we call Algorithm 9 when using KS3 (or KS4). That is, per call to Algorithm 9, $2L' - 3$ bit shifts to the right by ℓ' to compute δ in line 14. On the other side, when using KS2 (or KS4), we have the additional overhead of 1 bit shift to the right by 1 and 1 bit shift to the right by $\ell + 1$ when computing Equation 4.1 and Equation 4.2, respectively.
Another thing we should keep in mind are the needed modular reductions that we already analysed in Subsection 5.1.6.

Proof-of-concept. We provide proof-of-concept implementations in `SageMath`[2], a free open-source mathematics software system, and `Python 3.7`. Our chosen hard-coded parameter set is sntrup761, but switching to the other recommended sets is as simple as changing the value definitions for p, q and t

[2]`https://www.sagemath.org/`

Method	**# of operations**		**Total # of operations**	**Change comp. to** (1)
	≤ 64**-bit**	256**-bit**		
KS1, $\ell = 32$	4563	9227	13790	-81%
KS2, $\ell = 16$	6086	4906	10992	-85%
KS3, $\ell = 16$	19770	4715	24485	-67%
KS4, $\ell = 8$	22812	2649	25461	-66%

Operation counts for $w_{copro} = 256$.

Method	**# of operations**		**Total # of operations**	**Change comp. to** (1)
	≤ 64**-bit**	512**-bit**		
KS1, $\ell = 32$	4563	2355	6918	-91%
KS2, $\ell = 16$	6086	1323	7409	-90%
KS3, $\ell = 16$	19770	1227	20997	-72%
KS4, $\ell = 8$	22812	759	23571	-68%

Operation counts for $w_{copro} = 512$.

Method	**# of operations**		**Total # of operations**	**Change comp. to** (1)
	≤ 64**-bit**	1024**-bit**		
KS1, $\ell = 32$	4563	613	5176	-93%
KS2, $\ell = 16$	6086	379	6465	-91%
KS3, $\ell = 16$	19770	331	20101	-73%
KS4, $\ell = 8$	22812	238	23050	-69%

Operation counts for $w_{copro} = 1024$.

Method	**# of operations**		**Total # of operations**	**Change comp. to** (1)
	≤ 64**-bit**	2048**-bit**		
KS1, $\ell = 32$	4563	166	4729	-94%
KS2, $\ell = 16$	6086	119	6205	-92%
KS3, $\ell = 16$	19770	95	19865	-73%
KS4, $\ell = 8$	22812	84	22896	-69%

Operation counts for $w_{copro} = 2048$.

(1): Total number of operations needed for the mixed-radix NTT reference implementation by [Alk+20], i.e. 74142 32-bit operations.

Table 6.3: Total number of operations that need to be performed for each Kronecker substitution variant. The different segments show different word sizes of the cryptographic co-processor w_{copro}, and the changes illustrate the total number of needed operations compared to those of (1).

in the code.
Our implementation includes KS1, KS2, KS3 and KS4 for both the unsigned and the signed case. The `Sage` reference implementation for the whole cryptosystem sntrup761 was taken from the official NTRU Prime website[3].
We provide two different variants of our implementation. The first is a typical `Python` program that uses all built-in functionality provided by the `SageMath` library (big-integer arithmetic, ring arithmetic, ...). The second implementation is more "`C`-like", in the sense that it stores big-integers in array-like structures and performs every operation "manually". This variant aims to prove our concepts of packing, unpacking, reduction in the ring, and big-integer arithmetic as it would need to be implemented within embedded languages.
We tested our implementations not only with standard data as generated in Streamlined NTRU Prime, but with extreme values, too. These are polynomials where all coefficients are equal to their maximal or minimal value (to test our calculated bounds from Section 5.1). To verify our calculated ℓ's as given in Table 5.2 in practice, we wrote a test that tries out every reasonable ℓ per setting and returns the smallest one for that the multiplication results were still correct. We also ran all tests provided in the aforementioned `Sage` reference implementation of sntrup761, where we replaced all polynomial multiplications by our different Kronecker substitution variants.

Important considerations. To describe our results and compare them to state-of-the-art implementations, we purposefully ignore certain aspects that the reader should be aware of. Although our results are very well defined, we want to note that we do not consider all possible implementation aspects, as this would be beyond the scope of this thesis.
Exemplary overhead includes loads and stores: the time needed to store the large integer in the co-processor's memory, and the time needed to load the result into CPU memory again. As these timings are very system specific, it was not possible to estimate them in a fair manner.
Another important aspect is the memory overhead, i.e. the exact numbers of additional memory the Kronecker substitution might need compared to other

[3]Official website: `https://ntruprime.cr.yp.to/software.html`. 2019 Reference implementation: `https://ntruprime.cr.yp.to/ntruprime.sage`

polynomial multiplication techniques. While we talk about memory consumption depending on different choices of ℓ and which variants of the Kronecker substitution need the least/most amount of memory for intermediate computations, this all highly depends on the native underlying hardware. As we chose to not depend on hardware too much, we leave the memory discussion on a very high level. However, we note that future work on specific systems should always include a thorough analysis of memory overhead.
As the last point, we mention cycle counts. On different systems, different operations take a different number of cycles to complete. If our suggestions are implemented on specific hardware, we recommend performing actual speed measurements that involve cycle counts to get a confident result.

7 Conclusion

With Streamlined NTRU Prime, we chose a very recent and promising post-quantum cryptosystem to analyse. During the work on this thesis, the family of cryptosystem NTRU Prime even advanced to an alternate candidate in the third and final round of the NIST PQC standardisation competition. NTRU Prime's success in the competition further underlines the significance of lattice-based systems when talking about post-quantum secure cryptography.
The polynomial multiplication, the main bottleneck of lattice-based cryptosystems, was an interesting point for further enhancements in this thesis. With Kronecker substitution, a method that converts polynomial arithmetic into large-integer arithmetic, we selected a promising enhancement strategy. We also took several other flavours of the Kronecker substitution into account.
We were able to combine the ring $\mathbb{Z}_q[x]/(x^p - x - 1)$ of NTRU Prime with the method of Kronecker substitution. This procedure involved a thorough bound study for the polynomial multiplication in the ring of NTRU Prime to conclude the correct evaluation points. Furthermore, we discussed some enhancements for the packing and unpacking phases of Kronecker substitution that are possible for our setting. For that, we compared different procedures to optimise the overhead that Kronecker substitution introduces.

Our results for Kronecker substitution and its variants are excellent if we assume co-processors for fast large-integer arithmetic. With an increasing assumed word size of the co-processor, the number of operations that need to be performed on such hardware decreases significantly. Most current state-of-the-art implementations for polynomial multiplications in NTRU Prime use Karatsuba's method, Toom-Cook multiplication, or even NTTs. That means they all perform many small-bit multiplications and additions to calculate the final product. In terms of operation counts, Kronecker substitution and its variants have unarguable advantages to those current implementations. We showed what an exciting alternative the Kronecker substitution is for embedded devices, also because such co-processors for fast large-bit integer arithmetic already exist. Instead of analysing enhancements for polynomial arithmetic on conventional computers, we could repurpose existing hardware built for asymmetric cryptosystems like RSA and ECC and utilise their integer arithmetic, similar to the work of [Alb+18].

The packing and unpacking steps constitute the main overhead of the Kronecker substitution on embedded devices. The small-bit operations that are involved in those are not influenced by the word size of the co-processor. A possible improvement is to pack the polynomials once at the start of the encapsulation or decapsulation phase and unpack them only at the very end of the respective function. While this is possible, it would undoubtedly involve some more big-integer operations on the packed values. Further enhancements of the packing and unpacking overhead would have gone beyond the scope of this thesis but should be further researched.

Another future task is applying our methods to other lattice-based cryptosystems. Doing so would involve a bound study in the respective ring of the chosen cryptosystem, but is otherwise straightforward. We expect that it is feasible to adapt our proof-of-concept implementations to other settings and that one can therefore very quickly test the compatibility with other systems. To get more hardware-specific results, one would need to write a careful implementation that is directly suited to the particular system, which goes beyond this thesis's scope.

Our results aim to set the theoretic basis for future practical work on embedded devices. We showed that Kronecker substitution is a straightforward way of shifting heavy computations to existing hardware and software solutions for large-integer arithmetic. Further, we proofed its applicability on the example NTRU Prime and provided very promising operation counts compared to state-of-the-art implementations of our target cryptosystem.

Bibliography

[Ajt98] M. Ajtai. "The Shortest Vector Problem in L_2 is NP-Hard for Randomized Reductions (Extended Abstract)". In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing.* STOC '98. Dallas, Texas, USA: Association for Computing Machinery, 1998, pp. 10–19. ISBN: 0897919629. DOI: `10.1145/276698.276705`. URL: `https://doi.org/10.1145/276698.276705` (cit. on p. 8).

[Alb+18] M. R. Albrecht, C. Hanser, A. Hoeller, T. Pöppelmann, F. Virdia, and A. Wallner. "Implementing RLWE-based Schemes Using an RSA Co-Processor". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019 (Nov. 2018), pp. 169–208. DOI: `10.13154/tches.v2019.i1.169-208`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/7338` (cit. on pp. 22, 23, 48, 67).

[Alk+16] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. "Post-Quantum Key Exchange: A New Hope". In: *Proceedings of the 25th USENIX Conference on Security Symposium.* SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 327–343. ISBN: 9781931971324 (cit. on p. 21).

[Alk+20] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wálde, and B.-Y. Yang. *Polynomial Multiplication in NTRU Prime: Comparison of Optimization Strategies on Cortex-M4.* Cryptology ePrint Archive, Report 2020/1216. `https://eprint.iacr.org/2020/1216`. 2020 (cit. on pp. 21, 22, 61, 63).

[Bel+98] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. "Relations Among Notions of Security for Public-Key Encryption Schemes". In: *Advances in Cryptology.* Springer Berlin Heidelberg, 1998, pp. 26–45 (cit. on p. 10).

[Ber+18] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. "NTRU Prime: Reducing Attack Surface at Low Cost". In: Jan. 2018, pp. 235–260. ISBN: 978-3-319-72564-2. DOI: `10.1007/978-3-319-72565-9_12` (cit. on pp. 3, 8, 10–12, 21, 39, 40).

[Ber+20] D. J. Bernstein, B. B. Brumley, M.-S. Chen, C. Chuengsatiansup, T. Lange, A. Marotzke, B.-Y. Peng, N. Tuveri, C. van Vredendaal, and B.-Y. Yang. *NTRU Prime: Round 3*. NIST Third-Round Submission. `https://ntruprime.cr.yp.to/nist/ntruprime-20201007.pdf`. 2020 (cit. on p. 11).

[Bos+18] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018, pp. 353–367. DOI: `10.1109/EuroSP.2018.00032` (cit. on p. 22).

[BRV20] J. W. Bos, J. Renes, and C. van Vredendaal. *Polynomial Multiplication with Contemporary Co-Processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer*. Cryptology ePrint Archive, Report 2020/1303. `https://eprint.iacr.org/2020/1303`. 2020 (cit. on pp. 51, 54, 55).

[BZ10] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010. DOI: `10.1017/CBO9780511921698` (cit. on pp. 12, 14–20).

[CCG00] E. Chu, E. Chu, and A. George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Computational Mathematics Series. CRC-Press, 2000. ISBN: 9780849302701. URL: `https://books.google.at/books?id=VjkZAQAAIAAJ` (cit. on p. 17).

[Che+20] H. Cheng, D. Dinu, J. Großschädl, P. B. Rønne, and P. Y. A. Ryan. "A Lightweight Implementation of NTRU Prime for the Post-quantum Internet of Things". In: *Information Security Theory and Practice*. Ed. by M. Laurent and T. Giannetsos. Cham: Springer International Publishing, 2020, pp. 103–119. ISBN: 978-3-030-41702-4 (cit. on p. 21).

[Coo66] S. A. Cook. "On the Minimum Computation Time of Functions". PhD thesis. Harvard University, 1966, pp. 51–77. URL: https://cr.yp.to/bib/1966/cook.html (cit. on pp. 16, 17).

[Din+03] I. Dinur, G. Kindler, R. Raz, and S. Safra. "Approximating CVP to Within Almost-Polynomial Factors is NP-Hard". In: *Combinatorica* 23 (Apr. 2003), pp. 205–243. DOI: 10.1007/s00493-003-0019-y (cit. on p. 8).

[Für07] M. Fürer. "Faster Integer Multiplication". In: *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing.* STOC '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 57–66. ISBN: 9781595936318. DOI: 10.1145/1250790.1250800. URL: https://doi.org/10.1145/1250790.1250800 (cit. on p. 19).

[Goo51] I. J. Good. "Random motion on a finite Abelian group". In: *Proceedings of the Cambridge Philosophical Society* 47.4 (Jan. 1951), p. 756. DOI: 10.1017/S0305004100027201 (cit. on p. 22).

[Har09] D. Harvey. "Faster Polynomial Multiplication via Multipoint Kronecker Substitution". In: *J. Symb. Comput.* 44.10 (Oct. 2009), pp. 1502–1510. ISSN: 0747-7171. DOI: 10.1016/j.jsc.2009.05.004. URL: https://doi.org/10.1016/j.jsc.2009.05.004 (cit. on pp. 20, 22, 24, 25, 27, 31, 33).

[HPS08] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography.* 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 0387779930 (cit. on pp. 5, 8, 10).

[HPS98] J. Hoffstein, J. Pipher, and J. H. Silverman. "NTRU: A Ring-Based Public Key Cryptosystem". In: *ANTS.* Ed. by J. Buhler. Vol. 1423. Lecture Notes in Computer Science. Springer, 1998, pp. 267–288. ISBN: 3-540-64657-4 (cit. on pp. 3, 7).

[KO63] A. Karatsuba and Y. P. Ofman. "Multiplication of Many-Digital Numbers by Automatic Computers (in Russian)". In: *Proceedings of the USSR Academy of Sciences.* Vol. 145. 1963, pp. 293–294 (cit. on p. 14).

[Kob87] N. Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (Jan. 1987), pp. 203–209. ISSN: 0025-5718. DOI: `10.1090/S0025-5718-1987-0866109-5`. URL: `https://doi.org/10.1090/S0025-5718-1987-0866109-5` (cit. on p. 48).

[Kob94] N. Koblitz. *A Course in Number Theory and Cryptography*. 2nd ed. Springer Publishing Company New York, 1994. ISBN: 978-1-4419-8592-7 (cit. on p. 5).

[Kro82] L. Kronecker. “Grundzüge einer arithmetischen Theorie der algebraischen Grössen. (Abdruck einer Festschrift zu Herrn E. E. Kummers Doctor-Jubiläum, 10. September 1881.).” ger. In: *Journal für die reine und angewandte Mathematik* 92 (1882), pp. 1–122. URL: `http://eudml.org/doc/148487` (cit. on p. 20).

[LPR12] V. Lyubashevsky, C. Peikert, and O. Regev. *On Ideal Lattices and Learning with Errors Over Rings*. Cryptology ePrint Archive, Report 2012/230. `https://eprint.iacr.org/2012/230`. 2012 (cit. on pp. 3, 22).

[LS15] A. Langlois and D. Stehlé. “Worst-Case to Average-Case Reductions for Module Lattices”. In: *Des. Codes Cryptography* 75.3 (June 2015), pp. 565–599. ISSN: 0925-1022. DOI: `10.1007/s10623-014-9938-4`. URL: `https://doi.org/10.1007/s10623-014-9938-4` (cit. on p. 3).

[LS19] V. Lyubashevsky and G. Seiler. “NTTRU: Truly Fast NTRU Using NTT”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.3 (May 2019), pp. 180–201. DOI: `10.13154/tches.v2019.i3.180-201`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/8293` (cit. on p. 3).

[Mao03] W. Mao. *Modern Cryptography: Theory and Practice*. Pearson India, 2003. ISBN: 978-8131702123 (cit. on p. 5).

[Mil85] V. S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology*. CRYPTO ’85. Berlin, Heidelberg: Springer-Verlag, 1985, pp. 417–426. ISBN: 3540164634 (cit. on p. 48).

[Nat] National Institute of Standards and Technology. *Post-Quantum Cryptography Round 2 Submissions*. URL: `https://csrc.nist.gov/Projects/post-quantum-cryptography` (cit. on p. 2).

[Reg09] O. Regev. “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”. In: *J. ACM* 56.6 (Sept. 2009). ISSN: 0004-5411. DOI: `10.1145/1568318.1568324`. URL: `https://doi.org/10.1145/1568318.1568324` (cit. on p. 3).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: `10.1145/359340.359342`. URL: `https://doi.org/10.1145/359340.359342` (cit. on pp. 22, 48).

[SS71] A. Schönhage and V. Strassen. “Schnelle Multiplikation großer Zahlen”. In: *Computing* 7 (1971), pp. 281–292. DOI: `10.1007/BF02242355`. URL: `https://doi.org/10.1007/BF02242355` (cit. on pp. 18, 19).

[Too63] A. L. Toom. “The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers (in Russian).” In: vol. 150(3). 1963, pp. 496–498. URL: `http://toomandre.com/my-articles/engmat/MULT-E.PDF` (cit. on p. 16).