EFFICIENT IMPLEMENTATION OF IDEAL LATTICE-BASED CRYPTOGRAPHY

DISSERTATION

---

for the degree of Doktor-Ingenieur
of the Faculty of Electrical Engineering and Information Technology
at the Ruhr-University Bochum, Germany

---

by Thomas Pöppelmann
Bochum, June 2015

Printed in Germany.

Thomas Pöppelmann
Place of birth: Oelde, Germany
Author's contact information:
`thomas.poeppelmann@rub.de`

Thesis Advisor: **Prof. Dr.-Ing. Tim Güneysu**
Ruhr-Universität Bochum, Germany
Secondary Referee: **Prof. Dr. Ir. Ingrid Verbauwhede**
K.U.Leuven, Belgium
Thesis submitted: June 16, 2015
Thesis defense: July 23, 2015
Last revision: June 12, 2016

Abstract

Digital signatures and public-key encryption are used to protect almost any secure communication channel on the Internet or between embedded devices. Currently, protocol designers and engineers usually rely on schemes that are either based on the factoring assumption (RSA) or on the hardness of the discrete logarithm problem (DSA/ECDSA). But in case of advances in classical cryptanalysis or progress on the development of quantum computers the hardness of these closely related problems might be seriously weakened. In order to prepare for such an event, research on alternatives is required to provide long-term security.

In this thesis, we focus on the efficient implementation of such alternative public-key cryptosystems whose security is based on the intractability of certain computational problems on ideal lattices. While an extensive theoretical background exists for lattice-based and ideal lattice-based cryptography, not much is known about the efficiency of practical instantiations, especially on constrained and cost-sensitive platforms. We thus investigate novel algorithms and implementation techniques for fast and flexible polynomial multiplication and Gaussian sampling and then use these building blocks to implement public-key encryption and signature schemes. The results provided in this thesis show that lattice-based schemes can be optimized for high performance or resource efficiency on embedded microcontrollers and reconfigurable hardware. Our implementations of a public-key encryption scheme based on the ring learning with errors problems (RLWE) or of the bimodal lattice signature scheme (BLISS) can even outperform classical ECC- and RSA-based implementations.

Lattice-based cryptography can also be used to realize homomorphic cryptography that allows computation on encrypted data. However, due to the large parameter sets and complex operations required, even for simple homomorphic evaluation operations, the performance of these schemes is a major issue preventing practical usage. In this thesis we investigate options for acceleration of homomorphic cryptography in a cloud environment using reconfigurable hardware. We implement all evaluation operations of the YASHE homomorphic encryption scheme and propose methods to deal with large ciphertext and key sizes as well as limited memory bandwidth.

Keywords

Post-quantum cryptography, public-key cryptosystem, embedded system, microcontroller, FPGA

Kurzfassung

Digitale Signaturen und Public-Key-Verschlüsselung werden für den Schutz nahezu jeder sicheren Kommunikation über das Internet oder zwischen eingebetteten Systemen genutzt. Die Sicherheit basiert dabei entweder auf der Faktorisierungsannahme (RSA) oder der Annahme, dass es schwer ist, das diskrete Logarithmus-Problem (DSA/ECDSA) zu lösen. Durch Fortschritte in der klassischen Kryptoanalyse oder bei der Entwicklung von Quantencomputern könnten diese Probleme allerdings in Zukunft ernsthaft geschwächt oder gelöst werden. Daher ist Forschung zu alternativen Public-Key-Kryptosystemen erforderlich, die in der Lage sind, auch in diesem Fall Langzeitsicherheit zu gewährleisten.

In der vorliegenden Arbeit wird die effiziente Implementierung von alternativen Public-Key-Kryptosystemen betrachtet, deren Sicherheit auf schwer lösbaren Problemen in Idealgittern basiert. Während in der Literatur bereits ein umfangreicher theoretischer Hintergrund zu Kryptographie basierend auf Gittern und Idealgittern vorhanden ist, ist die Effizienz solcher Konstruktionen, insbesondere auf eingeschränkten und kostensensitiven Plattformen, noch nicht ausreichend erforscht. In dieser Arbeit werden daher neuartige Algorithmen und Implementierungstechniken für schnelle und flexible Polynommultiplikation und das Erzeugen von diskret normalverteilten Polynomen diskutiert. Diese Bausteine werden anschließend genutzt, um Public-Key-Verschlüsselungs- und digitale Signaturverfahren zu implementieren. Im Ergebnis kann nachgewiesen werden, dass gitterbasierte Kryptosysteme hohe Leistung erreichen und effizient auf Mikrocontrollern und rekonfigurierbarer Hardware realisiert werden können. Implementierungen eines Public-Key-Verschlüsselungsverfahrens basierend auf dem Ring Learning With Errors (RLWE) Problem oder des gitterbasierten Signaturschemas BLISS übertreffen sogar die Ausführungsgeschwindigkeit von klassischen ECC- und RSA-basierten Implementierungen.

Gitterbasierte Kryptographie kann auch verwendet werden, um homomorphe Verschlüsselungsverfahren zu konstruieren, die Rechenoperationen auf verschlüsselten Daten ermöglichen. Allerdings ist aufgrund der großen Parametersätze und des hohen Rechenaufwands selbst für einfache Operationen die praktische Anwendbarkeit derzeit unklar. In dieser Arbeit werden daher Möglichkeiten betrachtet, um homomorphe Verschlüsselungsverfahren in einer Cloud-Umgebung mit Hilfe von rekonfigurierbarer Hardware zu beschleunigen. Die vorgestellte Implementierung des homomorphen Verschlüsselungsverfahrens YASHE erlaubt die Ausführung aller Operationen, die für die Berechnung auf verschlüsselten Daten benötigt werden. Ein besonderer Fokus liegt auf der Entwicklung neuer Konzepte, um mit den großen Geheimtexten und Schlüsseln sowie begrenzter Speicherbandbreite umzugehen.

Schlagworte

Post-Quantum-Kryptographie, Public-Key-Kryptosystem, Eingebettetes System, Mikrocontroller, FPGA

Acknowledgements

This thesis would not have been possible without continuous support by my parents, my brother, friends, colleagues, and co-authors and I would like to say thank you.

Especially, I would like to thank my advisor Tim Güneysu for accepting me as a doctoral student, introducing me to the world of secure hardware, and for providing me with the necessary freedom and funding to pursue my research. Working and studying in the Hardware Security Group (sometimes also known as Secure Hardware Group) was a great pleasure and I was able to learn a lot. Special thanks also to my colleagues Alexander, Ingo, Oliver, Pascal, Stefan, and Tobias who always helped me out and supported me. I would also like to thank Christof Paar and all colleagues from EMSEC and I will always recall the Monday morning group breakfasts at 11am, traveling to sunny conference locations, social events, and after-work Glühwein. A special thank you goes also to Irmgard for helping with administrative tasks and being the good soul of the group as well as to Horst who runs a tight ship in his IT department and who always provided the best hardware and software to get the job done.

Additionally, I would like to thank all hard-working students I was able to supervise or work with, especially I would like to thank Tobias, Fabian, and Friedrich. I am also thankful to Kristin Lauter, Michael Naehrig, and Andrew Putnam for providing me the opportunity to do an internship at Microsoft Research in Redmond. My research would also not have been possible without teaming up with co-authors and academic collaborators and thus I would like to thank you. I would also like to thank Ingrid Verbauwhede for agreeing to be my second referee. Moreover, I am thankful to everybody who volunteered to proof read some parts of this work.

Table of Contents

Chapter 1

Introduction

In this chapter we briefly introduce the problem of missing diversity in practical public-key cryptography and the threat to RSA and ECC posed by a quantum computer. Moreover, we shortly discuss why lattice-based cryptography is a viable alternative to commonly used asymmetric encryption and signature schemes. Additionally, we shortly detail the structure of this thesis and summarize the presented research contributions regarding building blocks, public-key encryption, signature schemes, and homomorphic encryption.

Contents of this Chapter

1.1 Motivation

Security has become a crucial aspect of many recent hardware and software systems, in particular for those being potentially exposed to attacks for the next 10 to 20 years. A major building block in such systems are cryptographic functions that can guarantee protection over the entire lifespan of a device, protocol, or system. Examples are long-lasting vehicular or avionic systems that cause high load for servers and infrastructure but that also require the cost and energy efficient implementation of cryptography on constrained and embedded devices. In vehicular communication infrastructures, for example, elliptic curve cryptography (ECC) using 224-bit and 256-bit NIST curves was recently standardized [Soc06]. The relatively large security parameters for ECC were considered a conservative choice to achieve long-term security.

However, in case one of the numerous attempts and approaches to build a quantum computer turns out to be successful, Shor's algorithm [Sho94] could be used to break RSA and ECC in polynomial time. Due the popularity of RSA and ECC this would affect almost any practical deployment of asymmetric cryptography and thus many servers, clients, but also embedded devices. While quantum computers are not ready, yet, significant resources are spent to boost their further development [Koe13, RG13] and IBM announced that they might become available within the next 15 years [Cha12]. In addition, recent breakthroughs in classical cryptanalysis [Jou13, BGJT14] have cast further doubts on the hardness of the discrete logarithm problem as almost-polynomial time algorithms for the problem in small-characteristic fields are now available. It is also worth mentioning that even though ECC or RSA are currently considered secure, future cryptanalysis or quantum computers could be used to break the encryption of stored communication data. Already, such data is very likely collected by government bodies and also by private organizations and even in two or three decades the decryption of medical records or trade secrets might still pose a big threat to the users of the current cryptographic systems.

These imminent threats have motivated the investigation of other fundamental problems upon which asymmetric cryptography can be based. The resulting cryptosystems that are executed on classical computers but for which currently no fast quantum algorithms are known, are usually referred to as *post-quantum*. And while quantum computers might still be several years away, it is important to note that it usually takes several decades before alternative schemes have been thoroughly tested, gained trust in the community, have been standardized, and adopted by industry. However, currently available post-quantum candidate schemes suffer from relatively large key length or are not as efficient as RSA or ECC (see [BBD08]). As a consequence, RSA and ECC are still widely used and more research efforts are required to offer practical and efficient substitutes.

One promising post-quantum alternative that could match the performance or key sizes of RSA or maybe even ECC is lattice-based cryptography. While lattices have already been used to construct cryptosystems for several decades (see GGH [GGH97] and NTRU [HPS98]) they have received a lot more attention since the introduction of the user-friendly LWE [Reg05] and RLWE [LPR10b] problems. In particular, lattice-based cryptosystems instantiated in polynomial rings that allow a security reduction to hard problems on *ideal* lattices, instead of random lattices, support the design of schemes that achieve practical public-key, ciphertext, and signature lengths. The downside of using ideal lattices is that the added structure, which is the reason for the short public key length, might also allow more efficient cryptanalysis exploiting this structure. However, so far no serious attacks are known that perform significantly better for ideal lattices when parameters and underlying rings are chosen conservatively and with care.

While arguments like security proofs, asymptotic efficiency, and small public keys seem very appealing, not much was known about the practical efficiency of ideal lattice-based cryptosystems when this research was started. The first works covering the implementation of ideal lattice-based cryptosystems appeared only in 2012 [GFS+12, GLP12]. Since then the performance of software implementations has been improved by several orders of magnitude and the area consumption of hardware implementations has been reduced by large factors. It is now possible to perform a serious comparison of lattice-based cryptoschemes with other post-quantum schemes, based on experimental results. Additionally, works on efficient implementation provide feedback to cryptographers how to build even more efficient schemes and show which building blocks require more attention or optimization.

Even though research on the practical implementation of ideal lattice-based cryptography is still in an early stage, some ideal lattice-based schemes discussed and implemented in this work come already close to the efficiency of well researched RSA and even ECC implementations (for some metrics). Our research thus gives further evidence that lattice-based cryptography is a serious post-quantum alternative. Additionally, we still see room for further improvement and thus even better performance or security properties. The same was true for RSA or ECC implementations, which got better and better over the years and got tuned and optimized for more use-cases and platforms.

1.2 Structure of this Thesis

This thesis is structured as follows:

Preliminaries In Chapter 2 we review mathematical notation, computational problems on lattices and ideal lattices, and efficient polynomial multiplication algorithms based on the number theoretic transform (NTT). In Chapter 3 we introduce a previously proposed cryptosystem (denoted RLWEenc) based on the ring learning with errors (RLWE) problem, the Güneysu, Lyubashevsky, Pöppelmann (GLP) signature scheme, the bimodal lattice-based signature schemes (BLISS), and a somewhat homomorphic encryption (SHE) scheme denoted as YASHE (yet another somewhat homomorphic encryption).

Polynomial multiplication on reconfigurable hardware In Chapter 4 we describe an implementation of a microcode engine that is implemented on reconfigurable hardware and which is the basis for some of our implementations of encryption and signature schemes. It is built on top of an NTT-based polynomial multiplier and offers a simple instruction set suitable to realize the polynomial arithmetic and random sampling operations required by ideal lattice-based cryptosystems.

Public-key encryption In Chapter 5 we propose a performance optimized implementation of RLWEenc, which is based on our microcode engine. Moreover, we discuss methods to implement RLWEenc with a small area footprint. An optimized implementation of the NTT on constrained 8-bit devices is discussed in Chapter 6 and used to accelerate a RLWEenc software implementation.

Signature schemes We describe our efficient implementations of the GLP and BLISS signature schemes on reconfigurable hardware in Chapter 7. Our results for software implementations of lattice-based signature schemes are covered in Chapter 8.

Somewhat homomorphic encryption A hardware design to accelerate the YASHE somewhat homomorphic encryption scheme (SHE) on a cloud platform is proposed in Chapter 9.

Future work To conclude the thesis we review the results and discuss future work like implementation security, the implementation of advanced schemes, as well as optimization of building blocks like the NTT.

1.3 Summary of Research Contributions

Most of the contribution of this thesis has already been published in conferences or journals. The works of the author on lattice-based cryptography that are the foundation of this thesis are [PG12, PG13, OPG14, PG14, PDG14a, PDG14b, GLP15, HPO+15, POG15a, POG15b, PNPM15a, PNPM15b]. Note that some papers are the basis for chapters or sections but that we sometimes also rearranged content for better readability. The contributions in [GLP12, GOPS13, DBG+14] are also related to lattice-based cryptography and some parts written by the author of this thesis are included. However, the main results of these papers are not part of the contribution and only used for comparison. The papers [BEE+12a, DGK+12, EDW+14] were co-authored during the time-frame of the author's doctoral project but are not part of the contribution and unrelated to lattice-based cryptography. The work published in [EHvM+10, LPR+10a, BNP+11] was carried out before the beginning of the author's doctoral project.

We would also like to note that due to the previous publication in conference proceedings some results of this thesis have already been improved by other researchers or have been used as basis for different optimizations or implementations on other platforms. In some cases we have incorporated these new results into our implementation and description but sometimes we also refer to the related work and detail the differences or improvements. Works that directly or indirectly rely on our research or improve results discussed in this work are, e.g., [APS13, MBFK14, CMV+14, RVM+14, AS15, dCRVV15].

The actual contribution of this thesis can be structured into four categories. The first category are general purpose building blocks applicable to a whole range of schemes. The second and third category are novel implementations of public-key encryption and signature schemes on various platforms. The fourth is an implementation of a homomorphic encryption scheme.

Building Blocks for Lattice-Based Cryptography

We provide building blocks that are applicable to a wide range of current ideal lattice-based cryptographic schemes and that are supposed to be relevant for future schemes as well. Our most versatile contribution is a microcode engine, implemented on reconfigurable hardware that supports NTT-based polynomial multiplication and storage of a configurable number of temporary polynomials (see Chapter 4). Moreover, we propose optimizations of the NTT on constrained devices and achieve high speed by efficiently moving permutations and preceding computations into the actual NTT algorithm (see Section 6.2). Additionally, we describe efficient uniform and Gaussian samplers required by lattice-based encryption and signature schemes. On a microcontroller we evaluate the Knuth-Yao, Ziggurat, and Bernoulli-based sampling algorithms (see Section 8.3.1). Furthermore, we evaluate the performance of an optimized discrete Gaussian sampler on reconfigurable hardware (see Section 7.3.1) as well as the ATxmega 8-bit microcontroller platform (see Section 8.4). The sampler was developed by Léo Ducas and is based on a convolution theorem on discrete Gaussians, the cumulative distribution table (CDT) approach, and Kullback-Leibler divergence.

Implementation of a Public-Key Encryption Scheme

By relying on the microcode engine as building block we propose a high performance implementation of RLWE-based public-key encryption (RLWEenc) on reconfigurable hardware. Due to the versatility of the microcode engine and decomposition of polynomial multiplications using the NTT, one core can support key generation, encryption, and decryption and achieves exceptional performance and relatively low area consumption (see Section 5.3). At the time the original work was published, it provided a significant improvement of the time-area product compared to previous work. Additionally, we propose a fast implementation of RLWEenc on an 8-bit ATxmega platform (see Chapter 6) that is using an optimized NTT implementation.

Implementation of Signature Schemes

We provide the first implementations of the GLP and BLISS signature schemes on reconfigurable hardware (see Chapter 7). The GLP core supports signing and verification in dual-mode and is using the NTT for fast polynomial multiplication. With our BLISS core we show that signature schemes that require Gaussian sampling can be implemented efficiently on reconfigurable hardware. For similar area consumption, BLISS signing is about five times faster than the GLP core and our core outperforms implementations of other post-quantum and classical schemes as well. It uses more efficient sparse multiplication, the KECCAK hash function, and Huffman encoding to reduce the final signature size close to its theoretical minimum. Moreover, we propose a performance improvement for a GLP software implementation and the first software implementation of BLISS on a Cortex-M4F microcontroller. Our implementation of BLISS on an ATxmega 8-bit microcontroller is several times faster than previous work.

Implementation of a Homomorphic Encryption Scheme

We provide the first fully functional FPGA implementation of the evaluation operations of the somewhat homomorphic encryption scheme YASHE (see Chapter 9). The implementation is developed and tested on an accelerator board optimized for the usage in cloud servers. To deal with the huge parameter sets required for meaningful homomorphic capabilities, we propose methods for efficient multiplication of large polynomials using a cache-friendly NTT algorithm. Due to external memory accesses being a significant bottleneck, we optimize the evaluation algorithms and the NTT computations for optimal usage of the DRAM's burst mode. Our results show that it is feasible to accelerate SHE on FPGAs despite of the complexity and size of the parameters.

Chapter 2

Background on Lattices, Polynomial Multiplication, and Gaussian Sampling

In this chapter we review mathematical notation and provide background information on lattices, ideal lattices, and related computationally hard problems. Additionally, we cover variants of the schoolbook and number theoretic transform (NTT) polynomial multiplication algorithms as well as algorithms that can be used to sample from a discrete Gaussian distribution. This chapter contains only preliminaries but no original work by the author of this thesis. The background regarding the NTT and polynomial multiplication is based on [PG12, GLP15] and some general content from [OPG14, HPO+15] is included. The description of the Bernoulli and convolution cumulative distribution table (CDT) samplers originally appeared in [PDG14a, PDG14b]. The approach and analysis of using a convolution of Gaussians in combination with the Kullback-Leibler divergence has been developed by Léo Ducas.

Contents of this Chapter

2.1 Introduction

Lattice-based cryptography has received much attention from cryptographers and a comprehensive background on hard problems and security reductions exists. However, the focus of this thesis is the practical implementation of lattice-based schemes. We thus only cover the background that is essential for the understanding of our work. For a general introduction to lattice problems we refer to the book by Micciancio and Goldwasser [MG02] and the chapter by Micciancio and Regev [MR09] in [BBD08]. We particularly focus on intermediate problems like the learning with errors (LWE) problem, the short integer solution problem (SIS), and the ring learning with errors problem (RLWE) as they appear in almost any lattice-based scheme. When constructing cryptosystems, these problems hide a large amount of the complexity of lattice-based cryptography.

For additional information on the mathematical foundations of ideal lattice-based cryptography we refer to the PhD theses of Lyubashevsky [Lyu08b], Xagawa [Xag10], Schneider [Sch11], Ducas [DB13], Lepoint [Lep14], and Langlois [Lan14] and to the courses by Micciancio [Mic14] and Regev [Reg09] for a treatment of theoretical foundations of modern lattice-based cryptography and cryptanalysis.

As the focus of this work is the efficient implementation of ideal lattices we also discuss algorithms for polynomial multiplication and discrete Gaussian sampling. When working with schemes based on ideal lattices, polynomial addition, polynomial subtraction, and polynomial multiplication are basic operations. A particularly useful tool to speed up polynomial multiplication is the number theoretic transform (NTT), which is basically a fast Fourier transform (FFT) over the integers modulo q [Nus82, Win96, GG03, Bla10] and also discussed in this chapter. Another major building block in a lot of lattice-based encryption and especially signature schemes is discrete Gaussian sampling, which can be expensive to implement on constrained devices for large standard deviations. Thus we review various algorithms that allow certain trade-offs and optimization for specific goals, e.g., performance, table size, or no dependency on floating point arithmetic (see [Fol14, DG14]).

2.2 Notation

In this thesis we use the following notation and definitions of vectors, matrices, polynomials, and distributions.

Sets

By the symbol $\mathbb{N}$ we denote the set of positive integers, by $\mathbb{Z}$ we denote the ring of integers, and by $\mathbb{Z}_q$ the ring $\mathbb{Z}/q\mathbb{Z}$ of integers modulo an integer q which is represented by elements in the interval $[-q/2, q/2) \cap \mathbb{Z}$. By the symbol $\mathbb{R}$ we denote the set of real numbers.

Vectors and Matrices

We denote column vectors by bold lower case letters (e.g., $\mathbf{v} = (v_1, \ldots, v_n)^T$ where $\mathbf{v}^T$ is the transpose) and matrices by bold upper case letters (e.g., $\mathbf{M}$). In case of matching dimensions of two vectors $\mathbf{x}, \mathbf{y}$, their inner product is denoted by $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_i x_i y_i$. The Euclidean norm of a vector (ℓ_2 norm) is defined as

$$\|\mathbf{x}\| = \sqrt{\langle \mathbf{x}, \mathbf{x} \rangle} = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

For the general case we define the ℓ_p norm for $p \in (0, \infty)$ as

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

and the infinity norm ℓ_∞ is defined as $\|\mathbf{x}\|_\infty = \max_{i=1}^{n} |x_i|$.

Polynomials

By $\mathbb{Z}[\mathbf{x}]$ and $\mathbb{R}[\mathbf{x}]$ we denote the sets of polynomials (in an indeterminate variable $\mathbf{x}$) with coefficients in $\mathbb{Z}$ and $\mathbb{R}$, respectively.

Definition 1 (Characteristics of Polynomials, statement from [Lyu08b])**.** *A polynomial is monic if the coefficient of the highest power is one. A polynomial in $\mathbb{Z}[\mathbf{x}]$ is irreducible if it cannot be represented by polynomials of lower degree that are also in $\mathbb{Z}[\mathbf{x}]$. A polynomial with degree less than n can be represented as a n-dimensional vector with the coefficients of the polynomial as coordinates.*

We further denote by $\mathbf{g}[i]$ the coefficient g_i of a polynomial $\mathbf{g} = g_0 + g_1\mathbf{x} + \cdots + g_{n-1}\mathbf{x}^{n-1}$. The product of two n-dimensional vectors $\mathbf{xy}$ is the $(2n-1)$-dimensional vector that is the product of the corresponding polynomials.

Definition 2 (The Ring $\mathcal{R}$, statement from [BLLN13b])**.** *Let d be a positive integer and define $\mathcal{R} = \mathbb{Z}[\mathbf{x}]/(\Phi_d(\mathbf{x}))$ as the ring of polynomials with integer coefficients modulo the d-th cyclotomic polynomial $\Phi_d(\mathbf{x}) \in \mathbb{Z}[\mathbf{x}]$. The degree of d is $n = \varphi(d)$, where φ is Euler's totient function. The elements of $\mathcal{R}$ can be uniquely represented by all polynomials in $\mathbb{Z}[\mathbf{x}]$ of degree less than n. Arithmetic in $\mathcal{R}$ is arithmetic modulo $\Phi_d(X)$, which is implicit whenever terms or equalities involving elements in R are used. An arbitrary element $\mathbf{a} \in \mathcal{R}$ can be written as $\mathbf{a} = \sum_{i=0}^{n-1} a_i\mathbf{x}^i$ with $a_i \in \mathbb{Z}$ and we identify $\mathbf{a}$ with its vector of coefficients $(a_0, a_1, \ldots, a_{n-1})$.*

When multiplying two polynomials $\mathbf{f}, \mathbf{g}$ in $\mathcal{R}$ we assume that the result $\mathbf{fg}$ is automatically reduced into $\mathcal{R}$. The object that is primarily used in this work is the ring $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ where all coefficients are automatically reduced modulo an integer modulus q into the interval $[-q/2, q/2)$. To denote the modulo q reduction of all coefficients of an element $\mathbf{a} \in \mathcal{R}$ we sometimes write $[\mathbf{a}]_q$.

Definition 3 (Invertibility of Polynomials in $\mathcal{R}$, statement from [BLLN13b])**.** *A polynomial $\mathbf{f} \in \mathcal{R}$ is invertible modulo q if there exists a polynomial $\mathbf{f}^{-1}$ such that $\mathbf{ff}^{-1} = \tilde{\mathbf{f}}$, where $\tilde{\mathbf{f}}(\mathbf{x}) = \sum_i a_i\mathbf{x}^i$ with $a_0 = 1 \bmod q$ and $a_j = 0 \bmod q$ for all $j \neq 0$.*

Sampling

By $a \xleftarrow{\$} S$ we denote the action of picking a independently and uniformly at random from some set S. For a finite S we sometimes denote the uniform distribution on S by $\mathcal{U}(S)$. Sometimes we also use the notation $\mathbf{a} \xleftarrow{\$} \mathcal{R}_{q,k}$ to denote the uniformly random sampling of $\mathbf{a} \in \mathcal{R}_q$ where all coefficients of $\mathbf{a}$ are $[-k, k]$. For a probability distribution χ on $\mathcal{R}$ we assume that we can sample efficiently and use the notation $\mathbf{a} \xleftarrow{\$} \chi$ to denote the random sampling of $\mathbf{a} \in \mathcal{R}$ from χ. The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ with mean 0 and standard deviation $\sigma > 0$ over the integers associates the probability $\rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$ to $x \in \mathbb{Z}$ for $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{x=1}^{\infty} \rho_\sigma(x)$. Thus by $d \xleftarrow{\$} D_{\mathbb{Z},\sigma}$ we denote the process of randomly sampling a value $d \in \mathbb{Z}$ according to $D_{\mathbb{Z},\sigma}$.

Miscellaneous

We define the logarithm $\log x$ to be base 2 unless otherwise stated. We use the standard Landau notation $\mathcal{O}$ so that $f(n) = \mathcal{O}(g(x))$ is defined as $\lim_{x\to\infty} \frac{f(x)}{g(x)} \neq \infty$. We use *iff* as a short form for if and only if.

2.3 Lattices and Ideal Lattices

A lattice is an arrangements of points in an Euclidean space with a regular structure.

Definition 4 (Lattice, statement from [Reg09, MR09])**.** *Given n linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^m$, the lattice generated by them is defined as*

$$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \left\{ \sum x_i \mathbf{b}_i | x_i \in \mathbb{Z} \right\}.$$

We refer to $\mathbf{b}_1, \ldots, \mathbf{b}_n$ as the basis of the lattice. We can equivalently define the basis $\mathbf{B}$ as the $m \times n$ matrix whose columns are $\mathbf{b}_1, \ldots, \mathbf{b}_n$ such that $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m\times n}$. The lattice that is generated by $\mathbf{B}$ is

$$\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \{\mathbf{Bx} | \mathbf{x} \in \mathbb{Z}^n\}.$$

The rank of the lattice is defined as n and its dimension is m. If $n = m$, the lattice is called a full-rank lattice.

Every point in the lattice can be described as linear combination of the linearly independent basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ by multiplying them with integer coefficients. The same lattice can be defined by different bases. Two bases are equivalent in case column vectors of the basis matrix are permuted, vectors of the basis matrix are negated, or vectors are added to integer multiples of other vectors. These operations can be described by the multiplication of the basis matrix $\mathbf{B}$ by any unimodular matrix $\mathbf{U}$ (i.e., $\det(\mathbf{U}) = \pm 1$). Thus, two bases $\mathbf{B}_1$ and $\mathbf{B}_2$ are equivalent if and only if $\mathbf{B}_2 = \mathbf{B}_1\mathbf{U}$, given a unimodular matrix $\mathbf{U}$ and two bases $\mathbf{B}_1$ and $\mathbf{B}_2$.

Definition 5 (Span of a Lattice, statement from [Reg09])**.** *The span of a lattice $\mathcal{L}(\mathbf{B})$ is the linear space spanned by its vectors,*

$$\operatorname{span}(\mathcal{L}(\mathbf{B})) = \operatorname{span}(\mathbf{B}) = \{\mathbf{By} | \mathbf{y} \in \mathbb{R}^n\}.$$

The length (ℓ_2 norm) of the shortest non-zero vector in a lattice, which equals the minimum distance, is denoted by λ_1. The generalization denoted as the i-th successive minima is defined such that the ball $\bar{\mathbf{B}}(0, r) = \{\mathbf{x} \in \mathbb{R}^m \,|\, \|\mathbf{x}\| \ \leq r\}$, of radius r and center $\mathbf{0}$, contains at least i linearly independent vectors [Mic11].

Definition 6 (Successive Minima, statement from [Reg09])**.** *Let Λ be a lattice of rank n. For $i \in \{1, \ldots, n\}$ we define the i-th successive minimum as*

$$\lambda_i(\Lambda) = \inf\{r | \dim(\operatorname{span}(\Lambda \cap \bar{\mathbf{B}}(\mathbf{0}, r))) \geq i\}$$

where $\bar{\mathbf{B}}(\mathbf{0}, r) = \{\mathbf{x} \in \mathbb{R}^m \,|\, \|\mathbf{x}\| \ \leq r\}$ is the closed ball of radius r around 0.

For more details on lattices we refer to the lecture notes of Regev [Reg09] and Micciancio [Mic14] and comprehensive works like [MG02, MR09, Mic11].

2.3.1 Computational Problems on Lattices

The shortest vector problem (SVP) and the closest vector problem (CVP) are two fundamental problems in lattices and their conjectured intractability is the foundation for a large number of cryptographic applications of lattices.

Definition 7 (The (Approximate) Shortest Vector Problem, statement from [Mic14], SVP)**.** *The shortest vector problem (*SVP*) asks, given a lattice basis* $\mathbf{B}$*, to find a shortest nonzero lattice vector, i.e., a vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ *with* $\|\mathbf{v}\| = \lambda_1(\mathcal{L}(\mathbf{B}))$*. In the* γ*-approximate* SVP_γ*, for* $\gamma \geq 1$*, the goal is to find a shortest nonzero lattice vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B}) \setminus \{\mathit{0}\}$ *of norm at most* $\|\mathbf{v}\| \leq \lambda_1(\mathcal{L}(\mathbf{B}))$*.*

The SVP asks for *a* shortest nonzero vector in a lattice, but not *the* shortest nonzero vector as several shortest vectors can exist. The approximate SVP_γ is more difficult for a small factor γ and becomes easier for an increasing γ. An algorithm that solves SVP in polynomial time and with exponential approximation factor $2^{\mathcal{O}(n)}$ is the Lenstra, Lenstra, Lovász (LLL) algorithm [LLL82], which was extended in works like [Sch87, SE94, GN08a] (see [NV09] for a survey). Algorithms that achieve an exact solution or approximate solutions of SVP within $\mathrm{poly}(n)$ factors, either run in $2^{\mathcal{O}(n)}$ and require exponential space [AKS01] or in $2^{\mathcal{O}(n \log n)}$ and require only polynomially space [Kan83]. Based on these observations Micciancio and Regev conclude that "there is no polynomial time algorithm that approximates lattice problems to within polynomial factors" [MR09].

Definition 8 (The (Approximate) Closest Vector Problem, CVP, statement from [Mic14])**.** *The closest vector problem (*CVP*) asks, given a lattice basis* $\mathbf{B}$ *and target vector* $\mathbf{t}$*, to find the lattice vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ *such that the distance to the target* $\|\mathbf{v} - \mathbf{t}\|$ *is minimized. In the* γ*-approximate* CVP_γ*, for* $\gamma \geq 1$*, the goal is to find a lattice vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ *such that* $\|\mathbf{v} - \mathbf{t}\| \leq \gamma \cdot \mathrm{dist}(\mathbf{t}, \mathcal{L}(\mathbf{B}))$ *where* $\mathrm{dist}(\mathbf{t}, \Lambda) = \inf\{\|\mathbf{v} - \mathbf{t}\| : \mathbf{v} \in \Lambda\}$ *is the distance of* $\mathbf{t}$ *to* Λ*.*

The CVP is the inhomogeneous version of SVP and can also be formulated as syndrome decoding problem for full rank lattices [Mic14].

For SVP NP-hardness was shown by van Emde Boas in [vEB81] for the ℓ_∞ norm. Ajtai then proved that SVP is NP-hard for the ℓ_2 norm using randomized reductions [Ajt98] where the corresponding decision problem is NP-complete. For CVP it was also shown in [vEB81] that the problem is NP-hard. However, when building cryptosystems in practice only subclasses of CVP or SVP are used that are not supposed to be NP-hard (see [HPS08, Remark 6.24.]). A comprehensive discussion of the hardness of SVP, CVP, and its variants can be found in [Xag10, Section 2.3] and [MG02]. In [HPS11] Hanrot et al. provide a survey on the history and state-of-the-art of solvers for SVP and CVP.

Definition 9 (The Approximate Shortest Independent Vectors Problem, SIVP, statement from [Mic14])**.** *The* γ*-approximate shortest independent vectors problem (*SIVP_γ*) asks, for* $\gamma \geq 1$ *and when given a basis* $\mathbf{B}$ *of an* n*-dimensional lattice, to find linearly independent vectors* $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathcal{L}(\mathbf{B})$ *such that* $\max_i \|\mathbf{v}_i\| \leq \gamma \lambda_n(\mathcal{L}(\mathbf{B}))$*.*

The SIVP asks for n short linearly independent vectors and is also provided as it plays an important role in the construction of cryptographic primitives [Mic11].

2.3.2 Average-Case Problems on Standard Lattices

For lattice-based cryptography two extremely popular average-case problems with a connection to hard lattice problems are the short (or small) integer solution (SIS) problem and the learning with errors (LWE) problem. These two problems (and their variants) appear to be extremely versatile and allow the construction of a variety of schemes, without explicit usage of lattices or standard lattice problems like the SVP or CVP. Note that we mostly adopt the notation and wording of Ducas [DB13] for the introduction of the following lattice problems and hardness results. For further information we provide a reference to the original work.

Definition 10 (The Short Integer Solution Problem [MR07], SIS)**.** *The short integer solution problem* $\mathrm{SIS}_{n,m,q,\beta}$ *with* m *unknowns,* $n \leq m$ *equations modulo* q *and norm-bound* β *is as follows: given a random matrix* $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ *chosen uniformly, find a non-zero short vector* $\mathbf{v} \in \mathbb{Z}_q^m \setminus \{\mathbf{0}\}$ *such that* $\mathbf{v}^T\mathbf{A} = \mathbf{0}^T$ *and* $\|\mathbf{v}\| \leq \beta$*.*

The SIS problem was first described by Ajtai [Ajt96] who proved that average-case one-way functions exist if certain lattice problems are hard in the worst-case. This result was improved by Micciancio and Regev in [MR07], who denoted the average-case problem as *small* integer solution problem. It is usually used to construct one-way functions, collision-resistant hash functions, or signature schemes and for appropriate parameters it is at least as hard as SIVP or a variant of SVP, the unique shortest vector problem (uSVP).

Theorem 1 (Hardness of SIS [Ajt96, MR04, GPV08], statement from [DB13])**.** *If there exists a probabilistic polynomial time (PPT) algorithm* $\mathcal{A}$ *that solves* $\mathrm{SIS}_{n,m,q,\beta}$ *instances with* $q \geq 2\beta\sqrt{n}$ *with non negligible probability then there exists a PPT algorithm* $\mathcal{B}$ *that solves* $\mathrm{uSVP}_{2\beta\sqrt{n}}$ *and* $\mathrm{SIVP}_{2\beta\sqrt{n}}$ *on any lattice of dimension* n*.*

The LWE problem was popularized by Regev [Reg05] who showed that, under a quantum reduction, solving a random LWE instance is as hard as solving certain worst-case instances of certain lattice problems. The LWE problem can be seen as a generalization of the learning parity with noise (LPN) problem [BFKL93] and is related to hard decoding problems [MR09]. In general, to solve the LWE problem, one has to recover a secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ when given a sequence of approximate random linear equations on $\mathbf{s}$. Non-quantum reductions from variants of the shortest vector problem to variants of the LWE problem have also been shown [Pei09]. The LWE problem is usually used to construct primitives such as CPA or CCA-secure public-key encryption, identity-based encryption (IBE), or fully-homomorphic encryption schemes [Reg10]. It can be defined as a search (sLWE) problem where the task is to recover the secret vector $\mathbf{s}$ or as a decision problem (dLWE) that asks to distinguish LWE samples from uniformly random samples.

Definition 11 (The Learning With Errors Problem [Reg05], search version, sLWE, statement from [DB13])**.** *The learning with errors problem, search version,* $\mathrm{sLWE}_{n,m,q,\chi}$*, with* n *unknowns,* $m \geq n$ *samples, modulo* q *and with error distribution* χ *is as follows: for a random secret* s *uniformly chosen in* $\mathbb{Z}_q^n$*, and given* m *samples of the form* $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \mod q)$ *where* $e \xleftarrow{\$} \chi$ *and* $\mathbf{a}$ *is uniform in* $\mathbb{Z}_q^n$*, recover the secret vector* $\mathbf{s}$*.*

Definition 12 (The Learning With Errors Problem [Reg05], decisional version, dLWE, statement from [DB13])**.** *The learning with errors problem, decisional version,* $\mathrm{dLWE}_{n,m,q,\chi}$*, with* n

unknowns, $m \geq n$ samples, modulo q and with error distribution χ is as follows: for a random secret $\mathbf{s}$ *uniformly chosen in $\mathbb{Z}_q^n$, and given m samples either all of the form* $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \mod q)$ *where $e \xleftarrow{\$} \chi$, or from the uniform distribution* $(\mathbf{a}, b) \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^n \times \mathbb{Z}_q)$*, decide if the samples come from the former or the latter case.*

An interesting property of the LWE problem is the equivalence of the (search) sLWE problem and the (decisional) dLWE problem. While it is clear that a solver for the sLWE problem can be used to solve the dLWE problem, it is also possible to solve the sLWE problem if the dLWE problem can be solved.

Theorem 2 (Decision to Search Reduction for LWE, statement from [DB13])**.** *For any integers n and m, any prime $q \leq poly(n)$, and any distribution χ over $\mathbb{Z}_q$, if there exists a PPT algorithm that solves* $\text{dLWE}_{n,m,q,\chi}$ *with non-negligible probability, then there exists a PPT algorithm that solves* $\text{sLWE}_{n,m',q,\chi}$ *for some $m' = m \cdot poly(n)$ with non-negligible probability.*

The LWE problem is a particular case of the bounded distance decoding (BDD) problem, which is a special version of the CVP. For a given basis $\mathbf{B}$ and a target vector $\mathbf{t}$ for which it holds that $\text{dist}(\mathbf{t}, \mathbf{B}) \leq \gamma\lambda_1(\mathbf{B})$, the γ-approximate BDD_γ problem asks to find a lattice vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ closest to $\mathbf{t}$ [LM09]. Thus the BDD problem is a variant of the CVP where the target vector is within a guaranteed distance of the lattice [HPS11]. LWE is then basically an instance of the BDD problem in the lattice.

Theorem 3 (Worst-case to Average-case Connection for LWE [Reg05], statement from [DB13])**.** *If there exists a PPT algorithm $\mathcal{A}$ that solves* $\text{LWE}_{n,m,q,\chi}$ *for $\chi = D_{\mathbb{Z},\alpha q}$ where $\alpha q > 2\sqrt{n}$ with non negligible probability, then there exists a quantum polynomial time algorithm $\mathcal{B}$ that solves* uSVP_γ *and* SIVP_γ *on any lattice (i.e., in the worst case) of dimension n where $\gamma = \mathcal{O}(n/\alpha)$.*

The reduction in Theorem 3 uses the original result by Regev who showed that solving LWE is at least as hard as solving the uSVP_γ and SIVP_γ problem using quantum algorithms and appropriate approximation factors. Subsequent classical reductions for LWE have been provided in works like [Pei09, BLP$^+$13]. The reduction by Regev also somehow explains the prevalence of the discrete Gaussian distribution in lattice-based cryptography as the original proof only holds if $\mathbf{e}$ is sampled from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ with standard deviation $\sigma = \alpha q$. Recently, the hardness of LWE was also shown for different small error distributions, e.g., uniformly random from $(0, 1)$, under the assumption that the number of samples is limited [MP13].

Definition 13 (LWE with Small Secret, dLWE', statement from [DB13])**.** *The learning with errors problem* $\text{dLWE'}_{n,m,q,\chi}$*, with n unknown, $m \geq n$ samples, modulo q and with error distribution χ is as follows: for a random secret* $\mathbf{s}$ *drawn from $-\chi^n$, and given m samples either all of the form* $(\mathbf{a}, b = \langle \mathbf{s}, \mathbf{a} \rangle + e \mod q)$ *where $e \xleftarrow{\$} \chi$, or from the uniform distribution* $(\mathbf{a}, b) \xleftarrow{\$} \mathcal{U}(\mathbb{Z}_q^n \times \mathbb{Z}_q)$*; decide if the samples come from the former or the latter case.*

For some cryptographic applications it is important that both the error as well as the secret of LWE samples are small, e.g., to limit noise growth in public-key encryption or homomorphic encryption. In [ACPS09] an algorithm is given that solves dLWE using an efficient algorithm to solve dLWE'. Thus the LWE problem is no easier for a secret $\mathbf{s}$ with coefficients drawn from the error distribution χ and not uniformly random in $\mathbb{Z}_q$.

2.3.3 Ring Variant of LWE

The biggest practical issue of lattice-based cryptography based on SIS and LWE are huge key sizes and also quite inefficient matrix-vector and matrix-matrix arithmetic. A more efficient alternative is to define LWE over polynomial rings [LPR10b]. While certain properties can be established for various rings, we restrict ourselves to the ring $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ where n is a power of two.

Definition 14 (The Search Ring-LWE Problem [LPR10b], RsLWE)**.** *Let $\mathcal{R}$ denote the ring $\mathcal{R} = \mathbb{Z}[\mathbf{x}]/\langle x^n + 1\rangle$ for n being an integer power of 2, and $\mathcal{R}_q = \mathcal{R}/(q\mathcal{R})$ for some integer q. The $\text{RdLWE}_{m,q,\chi}$ problem, for $m \geq 1$ samples, modulo q and with error distribution χ over $\mathbb{Z}$ is defined as follows: for a secret $\mathbf{s} \in \mathcal{R}_q$ and $\mathbf{a} \in \mathcal{R}_q$, both with uniform coefficients in $\mathbb{Z}_q$, and given m samples of the form $(\mathbf{a}, \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e})$ where $\mathbf{e} \in \mathcal{R}_q$ has coefficients drawn from χ, recover the secret polynomial $\mathbf{s}$.*

Definition 15 (The Decisional Ring-LWE Problem [LPR10b], RdLWE, statement from [DB13])**.** *Let $\mathcal{R}$ denote the ring $\mathcal{R} = \mathbb{Z}[\mathbf{x}]/\langle x^n + 1\rangle$ for n being an integer power of 2, and $\mathcal{R}_q = \mathcal{R}/(q\mathcal{R})$ for some integer q. The $\text{RdLWE}_{m,q,\chi}$ problem, for $m \geq 1$ samples, modulo q and with error distribution χ over $\mathbb{Z}$ is defined as follows: for a secret $\mathbf{s} \in \mathcal{R}_q$ and $\mathbf{a} \in \mathcal{R}_q$, both with uniform coefficients in $\mathbb{Z}_q$, and given m samples either all of the form $(\mathbf{a}, \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e})$ where $\mathbf{e} \in \mathcal{R}_q$ has coefficients drawn from χ, or from the uniform distribution $(\mathbf{a}, \mathbf{b}) \overset{\$}{\leftarrow} \mathcal{U}(\mathcal{R}^2)$; decide if the samples come from the former or the latter case.*

The RsLWE and RdLWE problems translate the LWE problem into the ring setting (see [LPR10b] for more details and proofs). The hardness of RsLWE and RdLWE is then based on the worst-case hardness of short-vector problems on *ideal* lattices, which are a subclass of standard lattices. A decision to search reduction, as given in Theorem 2, was also shown for RsLWE and RdLWE but places certain restrictions on the underlying rings. Analog to Definition 13, it is also possible to sample the secret $\mathbf{s}$ from the error distribution χ instead of the uniform distribution $\mathcal{U}(\mathbb{Z}_q)$. This form of the RLWE (and also standard LWE) problem is sometimes called the "Hermite normal form". In the remaining parts of the thesis we will use the RLWE either in its Hermite or standard form.

2.4 Polynomial Arithmetic

Most cryptographic schemes based on ideal lattices that target practical application scenarios are instantiated using polynomial rings $\mathcal{R}_q = \mathcal{R}/(q\mathcal{R}) = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ for integer n and integer q (see Chapter 3).[1] In this case the basic operations required are polynomial addition and polynomial multiplication. While polynomial addition can be performed with $\mathcal{O}(n)$ operations in $\mathbb{Z}_q$, a naive implementation of polynomial multiplication requires $\mathcal{O}(n^2)$ operations in $\mathbb{Z}_q$ and can thus become very expensive for large dimensions n. In this section we thus shortly revisit well-known polynomial multiplication algorithms that are important for the implementations of ideal lattice-based cryptosystems discussed in this thesis.

---

[1] In almost all cases q is also a prime and n a power of two.

2.4.1 Schoolbook Multiplication

The product $\mathbf{c} = \mathbf{a}{\cdot}\mathbf{b}$ of polynomials $\mathbf{c}, \mathbf{a}, \mathbf{b} \in \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ can be computed by considering the special rule that $\mathbf{x}^n \equiv -1$. This leads to

$$\mathbf{a}{\cdot}\mathbf{b} = \left[\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\mathbf{a}[i]\mathbf{b}[j]\mathbf{x}^{i+j}\right] \bmod \langle \mathbf{x}^n + 1\rangle = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(-1)^{\left\lfloor \frac{i+j}{n}\right\rfloor}\mathbf{a}[i]\mathbf{b}[j]\mathbf{x}^{i+j \bmod n} \tag{2.1}$$

where each coefficient is implicitly reduced modulo q. Note that reduction modulo n in Equation 2.1 can be performed efficiently by using a bit mask when n is a power of two and $(-1)^{\left\lfloor \frac{i+j}{n}\right\rfloor}$ is 1 for $i + j < n$ and -1 otherwise (note that $i + j \leq 2n - 2$).

In practice two basic methods exist to implement schoolbook multiplication (for certain hybrid approaches we refer to [GPW$^+$04]). One is row-wise multiplication (alternatively called "operand scanning" multiplication), which is also used in Equation 2.1 and where one coefficient of an operand is fixed while it is multiplied by all coefficients of the second operand. Thus a complete row of partial products is added to the result. The second approach is column-wise multiplication (alternatively called "product scanning" or "Comba" [Com90] multiplication)[2] where a full column of the result is computed before moving on to the next column. Both methods have a complexity of $\mathcal{O}(n^2)$ operations in $\mathbb{Z}_q$ and require n^2 modular multiplications and $(n - 1)^2$ additions or subtractions.

2.4.2 The Number Theoretic Transform

Polynomial multiplication can be performed with $\mathcal{O}(n \log n)$ operations in $\mathbb{Z}_q$ using the convolution property of the number theoretic transform (NTT) [Pol71, Nus82, Win96, GG03, Bla10]. The NTT is basically a fast Fourier transformation (FFT) defined over a finite field or ring so that no inaccurate floating point or complex arithmetic is needed. Thus all complex roots of unity of the FFT are exchanged for integer roots of unity and computations are carried out in the ring of integers modulo an integer q. Main applications of the NTT, besides ideal lattice-based cryptography, are integer multiplication (e.g., Schönhage and Strassen [SS71]) and signal processing [Bla10]. As we want to use the NTT for polynomial multiplication we are only interested in NTTs with parameters that support the so-called circular convolution property (see Definition 18 and [Nus82]). Additionally, in this work we assume for simplicity that the modulus q is prime. For more details on the NTT with composite moduli we refer to [Nus82] and first introduce the definition of primitive roots of unity.

Definition 16 (Primitive Root of Unity, statement from [GG03])**.** *Let $\mathcal{R}$ be a ring, $n \in \mathbb{N}_{\geq 1}$, and $\omega \in \mathcal{R}$. The value ω is an n-th root of unity if $\omega^n = 1$. The value ω is a primitive n-th root of unity (or root of unity of order n) if it is an n-th root of unity, $n \in \mathcal{R}$ is a unit in $\mathcal{R}$, and $\omega^{n/t} - 1$ is not a zero divisor for any prime divisor t of n.*

Definition 17 (NTT Supporting Circular Convolutions, see [Nus82])**.** *For a given primitive n-th root of unity ω in $\mathbb{Z}_q$, where q is a prime, the generic forward NTT that is supporting circular*

---

[2] See [HS15] for a short discussion on the history of the column-wise multiplication method.

convolutions and transformation of a length-n vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$ *to* $\tilde{\mathbf{a}} = (\tilde{a}_0, \ldots, \tilde{a}_{n-1})$ *with elements in* $\mathbb{Z}_q$*, denoted as* $\tilde{\mathbf{a}} = \mathsf{NTT}(\mathbf{a})$*, is defined as*

$$\tilde{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod q, \;\; i = 0, 1, \ldots, n-1 \tag{2.2}$$

with the inverse transformation, denoted as $\mathbf{a} = \mathsf{INTT}(\tilde{\mathbf{a}})$*, is defined as*

$$a_i = n^{-1} \sum_{j=0}^{n-1} \tilde{a}_j \omega^{-ij} \bmod q, \;\; i = 0, 1, \ldots, n-1. \tag{2.3}$$

As q is required to be prime it is always ensured that $n^{-1} \bmod q$ and $\omega^{-1} \bmod q$ exist and thus the only remaining requirement for the existence of the NTT is the existence of a primitive root of unity ω.

Theorem 4 (Existence of the NTT Supporting Circular Convolutions, see [Nus82])**.** *An NTT of length n that supports circular convolutions and that is defined modulo a prime q exists if and only if* $n|(q-1)$*.*

As previously stated, we are only interested in NTTs supporting circular convolutions to use them for polynomial multiplication. The definition of the positive wrapped convolution and negative wrapped convolution is thus provided below.

Definition 18 (Circular Convolutions, statement from [Win96])**.** *Let* $\mathbf{a} = (a_0, \ldots, a_{n-1})$*,* $\mathbf{b} = (b_0, \ldots, b_{n-1})$ *be vectors over* $\mathbb{Z}$*. The convolution of* $\mathbf{a}$ *and* $\mathbf{b}$*, written as* $\mathbf{a} \odot \mathbf{b}$*, is the vector* $\mathbf{c} = (c_0, \ldots, c_{2n-1})$*, with*

$$c_i = \sum_{j=0}^{n-1} a_j b_{i-j}$$

where $a_k = b_k = 0$ *for* $k < 0$ *or* $k \geq n$*.*
The positive wrapped convolution of $\mathbf{a}$ *and* $\mathbf{b}$ *is the vector* $\mathbf{c} = (c_0, \ldots, c_{n-1})$ *with*

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

The negative wrapped convolution of $\mathbf{a}$ *and* $\mathbf{b}$ *is the vector* $\mathbf{c} = (c_0, \ldots, c_{n-1})$ *with*

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}.$$

These convolutions are highly related to polynomial multiplication. The multiplication of two degree-n polynomials can be seen as the convolution of their coefficient vectors. Additionally, the multiplication of two degree-n polynomials in $\mathbb{Z}[\mathbf{x}]/\langle \mathbf{x}^n - 1 \rangle$ corresponds to the positive wrapped convolution of their coefficient vectors and the multiplication of two degree-n polynomials in $\mathbb{Z}[\mathbf{x}]/\langle \mathbf{x}^n + 1 \rangle$ corresponds to the negative wrapped convolution of their coefficient vectors. Thus an efficient algorithm to compute convolutions yields an efficient algorithm for polynomial multiplication.

Theorem 5 (Convolution Theorem, see [Win96]). *Let ω be a primitive $2n$-th root of unity in $\mathbb{Z}_q$. Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})$ be vectors of length n with elements in $\mathbb{Z}_q$ and $\mathbf{a}' = (a_0, \ldots, a_{n-1}, 0, \ldots, 0)$, $\mathbf{b}' = (b_0, \ldots, b_{n-1}, 0, \ldots, 0)$ be the corresponding vectors of length $2n$, where the trailing components have been filled with zeros. With $\circ$ meaning component-wise multiplication then $\mathbf{a} \odot \mathbf{b} = \mathsf{INTT}(\mathsf{NTT}(\mathbf{a}') \circ \mathsf{NTT}(\mathbf{b}'))$.*

Using Theorem 5 it is now possible to multiply arbitrary polynomials. The two input polynomials are first mapped into a zero padded vector, transformed into the frequency domain, multiplied coefficient-wise, and then transformed back into the time domain. If the polynomial multiplication is performed in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$, the convolution theorem can also be used as the $2n$-degree result can be reduced modulo $\langle \mathbf{x}^n + 1\rangle$ afterwards. However, this is not optimal as by appending n zeros, the length of the input sequence to the transform doubles.

To remove the need for zero padding, we first, for $\mathbf{a}, \bar{\mathbf{a}} \in \mathcal{R}_q$ and $\psi \in \mathbb{Z}_q$, define $\bar{\mathbf{a}} = \mathsf{PowMul}_{\psi}(\mathbf{a}) = (a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1})$ as well as the inverse multiplication by powers of ψ^{-1} denoted as $\mathbf{a} = \mathsf{PowMul}_{\psi^{-1}}(\bar{\mathbf{a}})$.

Theorem 6 (Wrapped Convolution, see [Win96, LMPR08]). *Let ω be a primitive n-th root of unity in $\mathbb{Z}_q$ and $\psi^2 = \omega$. Let $\mathbf{a} = (a_0, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, \ldots, b_{n-1})$ be vectors of length n with elements in $\mathbb{Z}_q$.*

(1) The positive wrapped convolution of $\mathbf{a}$ and $\mathbf{b}$ is $\mathsf{INTT}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$.

(2) Let $\mathbf{d} = (d_0, \ldots, d_{n-1})$ be the negative wrapped convolution of $\mathbf{a}$ and $\mathbf{b}$. Let $\bar{\mathbf{a}} = \mathsf{PowMul}_{\psi}(\mathbf{a}), \bar{\mathbf{b}} = \mathsf{PowMul}_{\psi}(\mathbf{b})$. It then holds that $\mathbf{d} = \mathsf{PowMul}_{\psi^{-1}}(\mathsf{INTT}(\mathsf{NTT}(\bar{\mathbf{a}}) \circ \mathsf{NTT}(\bar{\mathbf{b}})))$.

By using Theorem 6 we can directly perform a negative wrapped convolution without zero padding. Additionally, we are getting the necessary reduction by the polynomial $\mathbf{x}^n + 1$ for "free" and can work with a transform length that is equal to the number of polynomial coefficients. The only restriction is that we have to find an n-th root of unity ω and its modular square root ψ such that $\psi^2 \equiv \omega \bmod q$. As a consequence, when q is a prime and n a power of two, the negative wrapped convolution approach is only possible in case $q \equiv 1 \bmod 2n$ [LMPR08].

2.4.3 Efficient Computation of the NTT

A straightforward computation of the NTT using Equation 2.2 would have quadratic complexity and would not be more efficient than the schoolbook approach. Thus, to realize fast polynomial multiplication using the convolution theorem a fast algorithm to compute the NTT is required. The most straightforward implementation of the NTT with $\mathcal{O}(n \log n)$ operations in $\mathbb{Z}_q$ is a Cooley-Tukey radix-2 decimation-in-time (DIT) algorithm [CT65].

The DIT NTT algorithm recursively splits the computation into a sub-problem on the even inputs and a sub-problem on the odd inputs of the NTT and an iterative description is given in Algorithm 1. It requires a so called bit-reversed reordering of the input polynomial (BitRev) to enable in-place computation of the NTT. Thus for $\mathbf{a}', \mathbf{a} \in \mathcal{R}_q$ we also define $\mathbf{a}' = \mathsf{BitRev}(\mathbf{a})$ so that each coefficient a'_i of $\mathbf{a}'$ contains the coefficient of $\mathbf{a}$ with its bitreversed index such that $a'_i = a_{\mathsf{BitrevInt}(i)}$ (see Algorithm 2 for $\mathsf{BitrevInt}$).

Algorithm 1 Fast Iterative Decimation-in-Time Number Theoretic Transform [CLRS09]

```
func $\mathsf{Fast\text{-}NTT}\omega, q^{\mathrm{DIT}}(\mathbf{a} \in \mathcal{R}_q)$
  $\mathbf{a} \leftarrow \mathsf{BitRev}(\mathbf{a})$
  $N \leftarrow n$
  $m \leftarrow 2$
  while $m \leq N$ do
    $s \leftarrow 0$
    while $s < N$ do
      for $i$ to $m/2 - 1$ do
        $N \leftarrow i \cdot n/m$
        $k \leftarrow s + i$
        $l \leftarrow s + i + m/2$
        $c \leftarrow \mathbf{a}[k]$
        $d \leftarrow \mathbf{a}[l]$
        $\mathbf{a}[k] \leftarrow c + \omega^N d \mod q$
        $\mathbf{a}[l] \leftarrow c - \omega^N d \mod q$
      end for
      $s \leftarrow s + m$
    end while
    $m \leftarrow m \cdot 2$
  end while
  return $\mathbf{a}$
end func
```

The algorithm applies the so-called Cooley-Tukey (CT) butterfly, which computes $c + \omega^N d$ and $c - \omega^N d$ for some values of $N \in \{0, n/2 - 1\}$ and $\omega, c, d \in \mathbb{Z}_q$ overall $\frac{n \log_2(n)}{2}$ times. It can be used to calculate the forward transformations $\mathsf{NTT}(\mathbf{a}) = \mathsf{Fast\text{-}NTT}(\mathbf{a})^{\mathrm{DIT}}_{\omega,q}$ as well as the inverse transformation as $\mathsf{INTT}(\mathbf{a}) = n^{-1}\mathsf{Fast\text{-}NTT}(\mathbf{a})^{\mathrm{DIT}}_{\omega^{-1},q}$.

The powers of the primitive root of unity ω (often referred to as *twiddle factors*) can be precomputed or the algorithm can be rearranged to minimize the number of exponentiations or multiplications to obtain them (see [RVM$^+$14]). When we only use NTT or INTT we refer to an implementation according to or very similar to Algorithm 1 (using a fast NTT algorithm). When it is clear from the context that a negacyclic convolution is computed we do not explicitly write PowMul_ψ or $\mathsf{PowMul}_{\psi^{-1}}$ before a call to the NTT algorithm.

Algorithm 2 Bit-Reversal of an Integer

```
func $\mathsf{BitrevInt}(x \in [0, 2^n))$
  $//x$ is an integer in binary form $x = x_{n-1} \ldots x_0$
  for $i = 0$ to $n - 1$ do
    $y_i \leftarrow x_{n-1-i}$
  end for
  return $y$
end func
```

Early works dealing with the FFT already covered methods to parallelize or to optimize the computation of the FFT/NTT on certain target architectures and a vast amount of research exists on the design of FFT algorithms (see [CG00, CP01] for a comprehensive treatment). We also refer to Chapter 6 where we discuss various optimizations of the NTT algorithms that specifically target a constrained microcontroller environment. In Chapter 9 we adapt the cached-FFT algorithm, which was proposed by Baas [Baa05, Baa99], to the NTT setting and use it for the implementation of a homomorphic encryption scheme. It is also worth mentioning that different flavors of the NTT like the Fermat and Mersenne variants exist [Nus82].

2.5 Discrete Gaussian Sampling over the Integers

The discrete Gaussian distribution plays an important role in lattice-based cryptography. For some cryptosystems that use standard random lattices, like GPV [GPV08], it is usually necessary to sample a point on a lattice $\mathcal{L} \in \mathbb{R}$ that is distributed according to a discrete Gaussian distribution. This expensive operation is usually mandated by the security proofs and worst-case reductions. However, the implementation of Gaussian sampling, especially on a lattice, is challenging and can lead to complex implementations. The sampling of polynomials according to a discrete Gaussian distribution is simpler and for polynomials in $\mathcal{R}_q$ where n is a power of two it is sufficient to sample each coefficient according to a discrete Gaussian distribution. Still, no computer running with finite time is able to sample from a perfect Gaussian distribution, so it is clear that sampled values can only approximate a Gaussian. However, the gap between the real Gaussian distribution and the sampled one has to be small, so that the security proofs hold.

In this section we thus first revisit the definition of the discrete Gaussian distribution, list general algorithms for discrete Gaussian sampling and then describe one approach designed for embedded systems and one approach using a convolution of Gaussians in more detail.

2.5.1 Definitions

Before we revisit practical sampling algorithms we provide some definitions regarding the discrete Gaussian distribution (see also Section 2.2). The continuous Gaussian distribution for a random variable E on $\mathbb{R}$, center $c \in \mathbb{R}$, and for $x \in \mathbb{R}$ is defined as $\Pr(E = x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-c)^2/(2\sigma^2)}$. However, it is not possible to obtain a discrete Gaussian distribution by simply rounding the output of a continuous Gaussian sampler.

Definition 19 (Discrete Gaussian Distribution, statement from [DG14])**.** *Let $\sigma \in \mathbb{R}_{>0}$ be fixed. The discrete normal distribution or discrete Gaussian distribution on $\mathbb{Z}$ with center $c \in \mathbb{R}$ and parameter σ is denoted $D_{\mathbb{Z},\sigma,c}$ and is defined as follows. Let*

$$S_{\sigma,c} = \sum_{k=-\infty}^{\infty} e^{\frac{-(k-c)^2}{2\sigma^2}} = 1 + 2\sum_{k=1}^{\infty} e^{\frac{-(k-c)^2}{2\sigma^2}}$$

and let E be the random variable on $\mathbb{Z}$ such that, for $x \in \mathbb{Z}, \Pr(E = x) = \rho_\sigma(x) = \frac{1}{S} e^{-(x-c)^2/(2\sigma^2)}$.

In some works (e.g., [LP11]) the *Gaussian parameter* s is used and the distribution is written as being proportional to $e^{-\pi x^2/(s^2)}$ for $s = \sqrt{2\pi}\sigma$. It is also possible to sample a lattice according to a discrete Gaussian distribution. For $\mathbf{x} \in \mathcal{L}$ and $\mathbf{c} \in \mathbb{R}^m$ the discrete Gaussian distribution

$D_{\mathcal{L},\sigma,\mathbf{c}}$ on $\mathcal{L}$ is given by $\Pr(\mathbf{x}) = \rho_{\mathcal{L},\sigma,\mathbf{c}}(\mathbf{x})/\rho_{\mathcal{L},\sigma,\mathbf{c}}(\mathcal{L})$ for $\rho_{\mathcal{L},\sigma,\mathbf{c}}(\mathbf{x}) = \exp(-\|\mathbf{x}-\mathbf{c}\|^2/(2\sigma)^2)$. However, in this thesis we only deal with lattice-based cryptography that requires sampling from a Gaussian distribution centered around 0 over the integers. Thus $D_{\mathbb{Z},\sigma}$ with mean 0 and standard deviation $\sigma > 0$ over the integers associates the probability $\rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$ to $x \in \mathbb{Z}$ for $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ and $\rho_\sigma(\mathbb{Z}) = 1 + 2\sum_{x=1}^{\infty} \rho_\sigma(x)$ (see [DG14] for more details).

A tail bound for discrete Gaussians is given in [Lyu11, Lemma 4.4(1)] and states that for any $k > 0$,

$$\Pr_{z \xleftarrow{\$} D_{\mathbb{Z},\sigma}} [|z| > k\sigma] \leq 2e^{\frac{-k^2}{2}}.$$

Thus, as also shown in [DG14], for $k = 12$

$$\Pr_{z \xleftarrow{\$} D_{\mathbb{Z},\sigma}} [|z| > 12\sigma] \leq 2e^{\frac{-12^2}{2}} < 2^{-100}.$$

When analyzing sampling algorithms it is helpful to be able to compare the required number of random bits to sample one value with a theoretical lower bound (see the discussion in [DG14]).

Definition 20 (Entropy of a Discrete Gaussians, statement from [DG14])**.** *The entropy of a sample from $D_{\mathbb{Z},\sigma}$ is*

$$H = -\sum_{k=-\infty}^{\infty} p_k \log_2(p_k)$$

for $p_k = \rho_\sigma(k)$.

2.5.2 Review of Algorithms for Discrete Gaussian Sampling

With the rising popularity of lattice-based cryptography and more focus on practical implementations, several algorithms for discrete Gaussian sampling have been proposed, which we shortly review in this section. For a detailed comparison of some algorithms, with focus on constrained architectures, we refer to work by Dwarakanath and Galbraith [DG14]. They state that discrete Gaussian sampling is challenging due to the need for a large amount of random numbers, the requirement for high precision floating point arithmetic or large tables, and quality requirements regarding statistical distance. An additional challenge in practice is certainly the protection against implementation attacks, especially attacks that exploit leakage of timing information. For further information we also refer to a survey on Gaussian sampling in lattice-based cryptography [Fol14] and to the PhD theses of Ducas [DB13] and Lepoint [Lep14] who dealt with Gaussian sampling in the context of lattice-based cryptography.

Rejection Sampling

One of the conceptually simplest algorithms to sample from a discrete Gaussian distribution is to choose a uniformly random $u \in \{-\tau\sigma, \ldots, \tau\sigma\}$ (in this case τ is denoted as tail-cut) and to accept it with a probability proportional to $\exp(-x^2/2\sigma^2)$ [DDLL13a, DG14]. However, this requires costly computation of the exp() function with high precision, a large number of random bits, and leads to ≈ 10 trials per sample.

Cumulative Distribution Table

The cumulative distribution table (CDT) requires a precomputed table to sample from a discrete Gaussian distribution. First a table M with entries $p_z = \Pr(x \leq z : x \leftarrow D_\sigma)$ for $z \in [0, \tau\sigma]$ has to be precomputed and stored with precision λ [Pei10, DDLL13a, DG14]. The necessary storage space is roughly $\tau\sigma\lambda$ bits and to save storage space commonly only $z \in [0, \tau\sigma]$ is stored. Sampling requires to choose a uniformly random y from the interval $[0, 1)$ and a bit b and return to return the integer $(-1)^b z \in \mathbb{Z}$ such that $y \in [p_{z-1}, p_z)$. Finding y in the table is just a search problem and either a linear or binary search algorithm or something more advanced can be used. Additionally, the CDT-table can contain a lot of zeros so that specific strategies can be used to further reduce the size of the table. Note that no floating point arithmetic is required as it is sufficient to just store the binary expansion of the fractional part instead of a float or double as all numbers used are smaller than 1.0.

Knuth-Yao Algorithm

The Knuth-Yao algorithm allows to sample from a discrete Gaussian distribution [RVV13,DG14] by constructing a binary tree from the probability matrix and by performing a random walk to sample an element. The probability matrix consists of the binary expansion of the probabilities of all $x \in [0, \tau\sigma]$ ignoring leading zero digits. The matrix determines a rooted binary tree with internal nodes that always have two successors, as well as terminal leaves. The leaves are labeled with the value that is returned if this leaf is reached during the random walk through the tree. The number of leaves at level n is equal to the number of 1's in column n of the probability matrix (starting with column 0). The row in which a one appears is used as label for one of the leaves. Then all remaining nodes become internal nodes with two successors that get labeled the same way.

Ziggurat Algorithm

The discrete Ziggurat algorithm [BCG+13] is an approach to optimize rejection sampling. For this purpose m rectangles with the left corners on the y-axis and the right corners on the graph of the probability distribution function are computed such that all rectangles have the same size. The entire area under the graph is then covered by rectangles and a rectangle R_i can efficiently be stored by just storing the coordinates (x_i, y_i) of the lower right corner.

To sample a value, first a rectangle R_i is uniformly random sampled. The next step is to uniformly choose a x value within the sampled rectangle. If this x value is smaller or equal to the x coordinate of the previous rectangle, x gets accepted, because all points $(x_j, y_j) \in R_i$ with $x_j \leq x_{i-1}$ definitively lie within the area covered by the graph. Otherwise, one has to sample a value y and compute the exp() function to determine whether a value gets rejected or accepted.

2.5.3 A Sampler Based on Bernoulli Trials

A sampler that is based on the usage of Bernoulli distributed variables was originally proposed in [DDLL13a]. A Bernoulli distributed variable $\mathcal{B}_c$ outputs one with a probability of $c \in [0, 1]$ and zero otherwise. Sampling from this distribution is easy and can be performed by lazily evaluating if $y < c$ for a uniformly random $y \in [0, 1)$ and precomputed c. The general idea of the sampler is to optimize rejection sampling by reducing the probability of rejections. This is done by sampling first from an intermediate distribution, called binary Gaussian distribution, and then from the target distribution. The rejection rate is thus reduced to ≈ 1.47 (compared to 10 for classical rejection sampling) and no computations of the exponentiation function exp() or large precomputed tables are necessary any more.

The first tool used in [DDLL13a] to construct the sampler is an algorithm to sample according to $\mathcal{B}_{\exp(-x/f)}$ for any positive integer x using $\log_2 x$ precomputed values as described in Algorithm 3. Next, this algorithm is used to transform a simple Gaussian distribution to a discrete Gaussian of arbitrary parameter σ. This simple Gaussian (called the binary Gaussian because the probability of each x is proportional to 2^{-x^2}) has parameter $\sigma_{\text{bin}} = \sqrt{1/(2\ln 2)} \approx 0.849$. It is straightforward to apply (Algorithm 4) because of the form of its (unnormalized) cumulative distribution

$$\rho_{\sigma_{\text{bin}}}(\{0,\ldots,j\}) = \sum_{i=0}^{j} 2^{-i^2} = 1.1001\underbrace{0\ldots0}_{4}1\underbrace{0\ldots0}_{6}1 \quad \ldots \quad \underbrace{0\ldots0}_{2(j-2)}1\underbrace{0\ldots0}_{2(j-1)}1\,.$$

From there, one easily builds the distribution $k \cdot D_{\mathbb{Z}^+,\sigma_{\text{bin}}} + \mathcal{U}(\{0\ldots k-1\})$ as an approximation of $D_{\mathbb{Z}^+,k\sigma_{\text{bin}}}$ which is corrected using rejection sampling technique (Algorithm 5). This rejection only requires variables of the form $\mathcal{B}_{\exp(-x/f)}$ for integers x. The last step is to extend the distribution from $\mathbb{Z}^+$ to the whole set of integers $\mathbb{Z}$ as described in Algorithm 6.

Algorithm 3 Sampling $\mathcal{B}_{\exp(-x/f)}$ for $x \in [0, 2^\ell)$

```
for $i = \ell - 1$ to $0$
    if $x_i = 1$ then
        sample $A_i \leftarrow \mathcal{B}_{c_i}$
        if $A_i = 0$ then return $0$
return $1$
```

2.5.4 A Sampler Based on a CDT and Gaussian Convolutions

Gaussian sampling using a large CDT has been shown to be an efficient strategy (see [DDLL13a]). In this section, we describe an enhanced CDT-based Gaussian sampler for use on constrained devices[3]. For simplicity, we explicitly refer to the parameter set of the BLISS-I signature scheme (see Section 3.6) that requires $\sigma = 215.73$. However, the result can be transferred to any other parameter set as well. To increase performance, we first analyze and improve the binary search step to reduce the number of comparisons. Secondly, we decrease the size of the precomputed tables. In Section 2.5.4 we therefore apply a convolution lemma for discrete Gaussians adapted from [Pei10] that enables the use of a sampler with much smaller standard deviation $\sigma' \approx \sigma/11$, reducing the table size by a factor 11. In Section 2.5.4 we finally reduce the size of the precomputed table further by roughly a factor of two using floating-point representation by introducing an *adaptive mantissa size*.

---

[3] The content of this section was taken from a joint publication with Tim Güneysu and Léo Ducas [PDG14b]. Note that the proofs and ideas are attributed to Léo Ducas. However, we include this section for easy reference and self-contained description of the implementation but do not consider the algorithmic design and proof of the properties of the sampler as original contribution of this thesis.

Algorithm 4 Sampling $D_{\mathbb{Z}^+,\sigma_{\mathrm{bin}}}$

```
func $D_{\mathbb{Z}^+,\sigma_{\mathrm{BIN}}}(x \in \mathbb{Z}^+$ from $D^+_{\sigma_{\mathrm{bin}}})$
  Generate a bit $b \leftarrow \mathcal{B}_{1/2}$
  if $b = 0$ then return 0
  for $i = 1$ to $\infty$ do
    draw random bits $b_1 \dots b_k$ for $k = 2i - 1$
    if $b_1 \dots b_{k-1} \neq 0 \dots 0$ then restart
    if $b_k = 0$ then return $i$
  end for
end func
```

Algorithm 5 Sampling $D_{\mathbb{Z}^+,k\sigma_{\mathrm{bin}}}$ for $k \in \mathbb{Z}$

```
sample $x \in \mathbb{Z}$ according to $D^+_{\sigma_{\mathrm{bin}}}$
sample $y \in \mathbb{Z}$ uniformly in $\{0, \dots, k-1\}$
$z \leftarrow kx + y$
sample $b \leftarrow \mathcal{B}_{\exp(-y(y+2kx)/(2\sigma^2))}$
if $\neg b$ then restart
return $z$
```

Algorithm 6 Sampling $D_{\mathbb{Z},k\sigma_{\mathrm{bin}}}$ for $k \in \mathbb{Z}$

```
Generate an integer $z \leftarrow D^+_{k\sigma_{\mathrm{bin}}}$
if $z = 0$ restart with probability $1/2$
Generate a bit $b \leftarrow \mathcal{B}_{1/2}$ and return $(-1)^b z$
```

Algorithm 7 Sampling $\mathcal{B}_a \oslash \mathcal{B}_b$

```
sample $A \leftarrow \mathcal{B}_a$; if $A$ then return 1
sample $B \leftarrow \mathcal{B}_b$; if $\neg B$ then return 0
restart
```

For those last two steps we require the "measure of distance"[4] for a distribution, called Kullback-Leibler divergence (KL) [KL51, CT91], which offers tighter proof than the usual statistical distance. Kullback-Leibler is a standard notion in information theory and already played a role in cryptography, mostly in the context of symmetric cryptanalysis [Vau03, BG09].

Binary Search with Shortcut Intervals

The CDT sampling algorithm uses a table $0 = T[0] \leq T[i] \leq \dots \leq T[S+1] = 1$ to sample from a uniform real $r \in [0, 1)$. The unique result x is obtained from a binary search satisfying that $T[x] \leq r < T[x+1]$ so that each output x has a probability $T[x+1] - T[x]$. For BLISS-I we need a table with $S = 2891 \approx 13.4\sigma$ entries to dismiss only a portion of the tail less than 2^{-128}. As a result, the naive binary search would require $C \in [\lfloor \log_2 S \rfloor, \lceil \log_2 S \rceil] = [11, 12]$ comparisons on average.

As an improvement we propose to combine the binary search with a hash map based on the first bits of r to narrow down the search interval in a first step (an idea that has already been described in [CA74, Dev86] as guide tables). For the given parameters and memory alignment reasons, we choose the first byte of r for this hash map: the unique $v \in \{0 \dots 255\}$ such that $v/256 \leq r < (v+1)/256$. This table I of intervals has length 256 and each entry $I[v]$ encodes the smallest interval (a_v, b_v) such that $T[a_v] \leq v/256$ and $T[b_v] \geq (v+1)/256$. With this approach, the search can be directly reduced to the interval (a_v, b_v). By letting C denote the number of comparison on average, we have that $\sum_v \frac{\lfloor \log_2(b_v - a_v) \rfloor}{256} \leq C \leq \sum_v \frac{\lceil \log_2(b_v - a_v) \rceil}{256}$. For this distribution this would give $C \in [1.3, 1.7]$ comparisons on average.

---

[4] Technically, Kullback-Leibler divergence is not a distance; it is not even symmetric.

Preliminaries on the Kullback-Leibler Divergence

We now present the notion of Kullback-Leibler (KL) divergence that is later used to further reduce the table size. Detailed proofs of following lemmata are given in [PDG14b].

Definition 21 (Kullback-Leibler Divergence)**.** *Let $\mathcal{P}$ and $\mathcal{Q}$ be two distributions over a common countable set Ω, and let $S \subset \Omega$ be the strict support of $\mathcal{P}$ ($\mathcal{P}(i) > 0$ iff $i \in S$). The Kullback-Leibler divergence, denoted as D_{KL} of $\mathcal{Q}$ from $\mathcal{P}$ is defined as:*

$$D_{KL}(\mathcal{P} \| \mathcal{Q}) = \sum_{i \in S} \ln \left(\frac{\mathcal{P}(i)}{\mathcal{Q}(i)} \right) \mathcal{P}(i)$$

with the convention that $\ln(x/0) = +\infty$ for any $x > 0$.

The Kullback-Leibler divergence shares many useful properties with the more usual notion of statistical distance. First, it is additive so that $D_{\mathrm{KL}}(\mathcal{P}_0 \times \mathcal{P}_1 \| \mathcal{Q}_0 \times \mathcal{Q}_1) = D_{\mathrm{KL}}(\mathcal{P}_0 \| \mathcal{Q}_0) + D_{\mathrm{KL}}(\mathcal{P}_1 \| \mathcal{Q}_1)$ and, second, non-increasing under any function $D_{\mathrm{KL}}(f(\mathcal{P}) \| f(\mathcal{Q})) \leq D_{\mathrm{KL}}(\mathcal{P} \| \mathcal{Q})$. An important difference though is that it is not symmetric. Choosing parameters so that the theoretical distribution $\mathcal{Q}$ is at KL-divergence about 2^{-128} from the actually sampled distribution $\mathcal{P}$, the next lemma will let us conclude the following[5]: if the ideal scheme $\mathcal{S}^{\mathcal{Q}}$ (i.e., BLISS with a perfect sampler) has about 128 bits of security, so has the implemented scheme $\mathcal{S}^{\mathcal{P}}$ (i.e., BLISS with our imperfect sampler).

Lemma 1 (Bounding Success Probability Variations)**.** *Let $\mathcal{E}^{\mathcal{P}}$ be an algorithm making at most q queries to an oracle sampling from a distribution $\mathcal{P}$ and returning a bit. Let $\epsilon \geq 0$, and $\mathcal{Q}$ be a distribution such that $D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq \epsilon$. Let x (resp. y) denote the probability that $\mathcal{E}^{\mathcal{P}}$ (resp. $\mathcal{E}^{\mathcal{Q}}$) outputs 1. Then, $|x - y| \leq \sqrt{q\epsilon/2}$.*

In certain cases, the KL-divergence can be as small as the square of the statistical distance. For example, noting $\mathcal{B}_c$ the Bernoulli variable that returns 1 with probability c, we have $D_{\mathrm{KL}}(\mathcal{B}_{\frac{1-\epsilon}{2}} \| \mathcal{B}_{\frac{1}{2}}) \approx \epsilon^2/2$. In such a case, one requires $q = O(1/\epsilon^2)$ samples to distinguish those two distributions with constant advantage. Hence, we yield higher security using KL-divergence than statistical distance for which the typical argument would only prove security up to $q = O(1/\epsilon)$ queries. Intuitively, statistical distance is the sum of absolute errors, while KL-divergence is about the sum of squared relative errors.

Lemma 2 (Kullback-Leibler divergence for bounded relative error)**.** *Let $\mathcal{P}$ and $\mathcal{Q}$ be two distributions of same countable support. Assume that for any $i \in S$, there exists some $\delta(i) \in (0, 1/4)$ such that we have the relative error bound $|\mathcal{P}(i) - \mathcal{Q}(i)| \leq \delta(i)\mathcal{P}(i)$. Then*

$$D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq 2 \sum_{i \in S} \delta(i)^2 \mathcal{P}(i).$$

Using floating-point representation, it seems now possible to halve the storage ensuring a relative precision of 64 bits instead of an absolute precision of 128 bits. Indeed, storing data

---

[5] Apply the lemma to an attacker with success probability 3/4 against $\mathcal{S}^{\mathcal{P}}$ and number of queries $< 2^{127}$ (amplifying success probability by repeating the attack if necessary), and deduce that it also succeeds against $\mathcal{S}^{\mathcal{Q}}$ with probability at least 1/4.

with slightly more than of relative 64 bits of precision (that is, mantissa of 64 bits in floating-point format) one can reasonably hope to obtain relative errors $\delta(i) \leq 2^{-64}$ resulting in a KL-divergence less than 2^{-128}. We further exploit this idea in Section 2.5.4. But first, we will also use KL-divergence to improve the convolution Lemma of Peikert [Pei10] and construct a sampler using convolutions.

Reducing Precomputed Data by Gaussian Convolution

Given that x_1, x_2 are variables from continuous Gaussian distributions with variances σ_1^2, σ_2^2, then their combination $x_1 + cx_2$ is Gaussian with variance $\sigma_1^2 + c^2\sigma_2^2$ for any c. While this is not generally the case for discrete Gaussians, there exists similar convolution properties under some smoothing condition as proved in [Pei10, MP13]. Yet those lemmata were designed with asymptotic security in mind; for practical purpose it is in fact possible to improve the $O(\epsilon)$ statistical distance bound to a $O(\epsilon^2)$ KL-divergence bound. We refer to [Pei10] for the formal definition of the smoothing parameter η; for our purpose it only matters that $\eta_\epsilon(\mathbb{Z}) \leq \sqrt{\ln(2 + 2/\epsilon)/\pi}$ and thus our adapted lemma allows to decrease the smoothing condition by a factor of about $\sqrt{2}$.

Lemma 3 (Adapted from Thm. 3.1 from [Pei10])**.** *Let* $x_1 \leftarrow D_{\mathbb{Z},\sigma_1}$*,* $x_2 \leftarrow D_{k\mathbb{Z},\sigma_2}$ *for some positive reals* σ_1, σ_2 *and let* $\sigma_3^{-2} = \sigma_1^{-2} + \sigma_2^{-2}$*, and* $\sigma^2 = \sigma_1^2 + \sigma_2^2$*. For any* $\epsilon \in (0, 1/2)$ *if* $\sigma_1 \geq \eta_\epsilon(\mathbb{Z})/\sqrt{2\pi}$ *and* $\sigma_3 \geq \eta_\epsilon(k\mathbb{Z})/\sqrt{2\pi}$*, then distribution* $\mathcal{P}$ *of* $x_1 + x_2$ *verifies*

$$D_{KL}(\mathcal{P} \| D_{\mathbb{Z},\sigma}) \leq 2\Big(1 - \Big(\frac{1+\epsilon}{1-\epsilon}\Big)^2\Big)^2 \approx 32\epsilon^2.$$

Remark. The factor $1/\sqrt{2\pi}$ in our version of this lemma is due to the fact that we use the standard deviation σ as the parameter of Gaussians and not the renormalized parameter $s = \sqrt{2\pi}\sigma$ often found in the literature.

To exploit this lemma, for BLISS-I we set $k = 11$, $\sigma' = \sigma/\sqrt{1+k^2} \approx 19.53$, and sample $x = x_1' + kx_2'$ for $x_1', x_2' \leftarrow D_{\mathbb{Z},\sigma'}$ (equivalently $k \cdot x_2' = x_2 \leftarrow D_{k\mathbb{Z},k\sigma'}$). The smoothness conditions are verified for $\epsilon = \sqrt{2^{-128}/32}$ and $\eta_\epsilon(\mathbb{Z}) \leq 3.860$[6]. Due to usage of the much smaller σ' instead of σ the size of the precomputation table reduces by a factor of about $k = 11$ at the price of sampling twice. However, the running time does not double in practice since the enhancement based on the shortcut intervals reduces the number of necessary comparisons to $C \in [0.22, 0.25]$ on average. For a majority of first bytes v the interval length $b_v - a_v$ is reduced to 1 and x is determined without any comparison.

Asymptotic cost. If one considers the asymptotic costs in σ, our methods allow one to sample using a table size of $\Theta(\sqrt{\sigma})$ rather than $\Theta(\sigma)$ by doubling the computation time. Actually, for much larger σ one could use $O(\log \sigma)$ samples of constant standard deviation and thus achieve a table size of $O(1)$ for computational cost in $O(\log \sigma)$.

CDT Sampling with Reduced Table Size

We recall that when doing floating-point error analysis, the relative error of a computed value v is defined as $|v - v_e|/v_e$ where v_e is the exact value that was meant to be computed. Using the

---

[6] In a previous version we stated $\eta_\epsilon(\mathbb{Z}) \leq 3.92$ which is not accurate and has been fixed in this version.

table $0 = T[0] \leq T[i] \leq \cdots \leq T[S+1] = 1$, the output of a CDT sampler follows the distribution $\mathcal{P}$ with $\mathcal{P}(i) = T[i+1] - T[i]$. When applying the results from KL-divergence obtained above, the relative error of $T[i+1] - T[i]$ might be significantly larger than the one of $T[i]$. This is particularly true for the tail, where $T[i] \approx 1$ but $\mathcal{P}(i)$ is very small. Intuitively, we would like the smallest probability to come first in the CDT. A simple workaround is to reverse the order of the table so that $1 = T[0] \geq T[i] \geq \cdots \geq T[S+1] = 0$ with a slight modification of the algorithm so that $\mathcal{P}(i) = T[i] - T[i+1]$. With this trick, the subtraction only increases the relative error by a factor of roughly σ. Indeed, leaving aside the details relative to discrete Gaussian, for $x \geq 0$ we have

$$\int_{y=x}^{\infty} \rho_s(y)dy \Big/ \rho_s(x) \leq \sigma \quad \text{whereas} \quad \int_{y=0}^{x} \rho_s(y)dy \Big/ \rho_s(x) \underset{x\to\infty}{\longrightarrow} +\infty.$$

The left term is an estimation of the relative-error blow-up induced by the subtraction with the CDT in the reverse order and the right term the same estimation for the CDT in the natural order. We aim to have a variable precision in the table $T[i]$ so that $\delta(i)^2\mathcal{P}(i)$ is about constant around $2^{-128}/|S|$ as suggested by Lemma 2 while $\delta(i)$ denotes the relative error $\delta(i) = |\mathcal{P}(i) - \mathcal{Q}(i)|/\mathcal{P}(i)$. As a trade-off between optimal variable precision and hardware efficiency, we propose the following data-structure. We define 9 tables $M_0 \dots M_8$ of bytes for the mantissa with respective lengths $\ell_0 \geq \ell_1 \geq \cdots \geq \ell_8$ and another byte table E for exponents, of length ℓ_0. The value $T[i]$ is defined as

$$T[i] = 256^{-E[i]} \cdot \sum_{k=0}^{8} 256^{-(k+1)} \cdot M_k[i]$$

where $M_k[i]$ is defined as 0 when the index is out of bound $i \geq \ell_k$. In other terms, the value of $T[i]$ is stored with $p(i) = 9 - \min\{k | \ell_k > i\}$ bytes of precisions. More precisely, lengths are defined as $[\ell_0, \dots, \ell_8] = [262, 262, 235, 223, 202, 180, 157, 125, 86]$ so that we store at least two bytes for each entry up to $i < 262$, three bytes up to $i < 213$ and so forth. Note that no actual computation is involved in constructing $T[i]$ following the plain CDT algorithm.

For evaluation, we used the closed formula for KL-divergence and measured $D_{\mathrm{KL}}(\mathcal{P}\|\mathcal{Q}) \leq 2^{-128}$. The storage required by the table is $2\ell_0 + \ell_1 + \cdots + \ell_8 \approx 2.0$ KB. The straightforward CDF approach requires each entry up to $i < 262$ to be stored with $128 + \log_2 \sigma$ bits of precisions and thus requires a total of at least 4.4 KB.

Chapter 3

Introduction to Practical Ideal Lattice-Based Cryptography

In this chapter we shortly revisit some basic definitions related to asymmetric cryptography and introduce post-quantum cryptography. We then recall a public-key encryption scheme that is based on the ring learning with errors problem (denoted RLWEenc*). Moreover, we describe the* GLP *and* BLISS *digital signature schemes and introduce the somewhat homomorphic encryption (SHE) scheme* YASHE*. All schemes that are discussed in this chapter have been previously designed and proposed by various authors and the chapter contains no original work done by the author of this thesis. However, for the background on* RLWEenc *some parts from [PG13, POG15a] were used. For the description of* GLP *we included parts of [GLP15, HPO+15] and for* BLISS *we used material from [PDG14a, PDG14b, OPG14, HPO+15]. The description of* YASHE *was taken from [PNPM15a].*

Contents of this Chapter

3.1 Introduction

In this thesis we focus on four ideal lattice-based, asymmetric, and post-quantum cryptosystems. Namely, a public-key encryption scheme denoted as RLWEenc [LPR10b, LP11], the Güneysu, Lyubashevsky, Pöppelmann (GLP) signature scheme, the bimodal lattice-based signature schemes (BLISS), and the somewhat homomorphic encryption scheme YASHE (yet another somewhat homomorphic encryption). The security of all four schemes is based on hard problems on ideal lattices and the schemes are defined in $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$. In most cases n is chosen as a power of two and q as a prime for which it holds that $q \equiv 1 \bmod 2n$ so that efficient algorithms for the computation of the NTT are available (see Section 2.4.2).

Additionally, we would like to refer to the lattice-based NTRU PKE (denoted as NTRUEncrypt) by Hoffstein, Pipher, and Silverman [HPS98] and to the NTRU signature scheme (denoted as NTRUSign) [HHP+03]. In this thesis we do not provide an implementation of original NTRU schemes and thus do not introduce them formally but we would like to note that schemes like BLISS, GLP, and YASHE heavily rely on ideas and assumptions related to NTRU. Moreover, NTRU predates ideal lattice-based cryptography and properly instantiated NTRUEncrypt instances have survived more than 15 years of cryptanalysis while NTRUSign is considered to be broken [NR09, DN12]. Therefore, NTRUEncrypt can be considered as a serious candidate post-quantum PKE scheme. However, as it has already been implemented on various architectures (see, e.g., [ABF+08, KY09, HVP10]) its implementation and optimization is not in the scope of this work. Also note that we focus on implementation aspects in this chapter and thus we only describe the computations that have to be performed but do not go into detail on the security proofs or the parameter selection process. For these information we refer to the original papers.

3.2 Post-Quantum Cryptography

Cryptography deals with the protection of information at rest or in transit in the presence of malicious third parties. A general introduction to all relevant aspects of cryptography and basic schemes and constructions like RSA, DSA, ECC, ECDSA, hash functions, and AES is provided in works like [KL07, HPS08, PP09, Gal12]. As we focus on asymmetric cryptography we shortly recall necessary notation on digital signatures and public-key encryption (PKE) schemes. Moreover, we introduce the concept of post-quantum cryptography.

3.2.1 Public-Key Cryptography

A PKE scheme is a tuple of probabilistic polynomial time (PPT) algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where $\mathsf{Gen}(1^n)$ takes as input the security parameter 1^n and outputs a keypair $(\mathsf{sk},\mathsf{pk})$ consisting of the secret key sk and the public key pk. To encrypt, $\mathsf{Enc}(\mu, \mathsf{pk})$ takes as input the public key pk and a message $\mu \in \mathcal{M}$, where $\mathcal{M}$ is the plaintext space, and outputs a ciphertext c. To decrypt, $\mathsf{Dec}(c, \mathsf{sk})$ takes as input the ciphertext c and the secret key sk and outputs the message $\mu \in \mathcal{M}$. In case of failure, $\mathsf{Dec}(c, \mathsf{sk})$ outputs $\perp$. A PKE scheme is correct if, for a negligible function negl, $\Pr[\mathsf{Dec}(\mathsf{Enc}(\mu, \mathsf{pk}), \mathsf{sk}) \neq \mu] \leq \mathsf{negl}(n)$. A basic security notion for a PKE is security against chosen plaintext attacks (CPA) or *semantic security*. In this model an adversary has access to the public key and an encryption oracle. To achieve CPA-security the adversary should not be able to distinguish between two ciphertexts that are returned by the oracle and which are encryptions of two messages of the adversary's choice. A more comprehensive security notion is security against chosen ciphertext attacks (CCA). In this model an attacker is basically given access to a decryption oracle for ciphertexts of his choice. This is a realistic scenario, e.g., considering a smart card, and the goal of the attacker is usually to extract the secret key or to decrypt a challenge ciphertext. For a more rigid definition of CPA and CCA security (and their variants) we refer to [KL07].

A digital signature scheme (DSS) is defined as a tuple of PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ where $\mathsf{Gen}(1^n)$ takes as input the security parameter 1^n and outputs a keypair $(\mathsf{sk},\mathsf{pk})$ consisting of the secret key sk and the public key pk. The signing algorithm $\mathsf{Sign}(\mu, \mathsf{sk})$ takes as input a message $\mu \in \mathcal{M}$ and the secret key sk and outputs a corresponding signature σ. The verification algorithm $\mathsf{Verify}(\mu, \sigma, \mathsf{pk})$ takes as input the message μ, the signature σ and the public key pk and outputs 1 if and only if (μ, σ) is a valid message/signature pair under pk, otherwise it outputs 0 (invalid). A signature scheme is correct if for every n, every $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^n)$, every $\mu \in \mathcal{M}$ it holds that $\mathsf{Verify}(\mu, \mathsf{Sign}(\mu, \mathsf{sk}), \mathsf{pk}) = 1$ (note that it would also be possible to have a definition with a negligible error as we gave for PKEs). An adversary breaks the signature scheme if he is able to produce a forgery. This means an adversary without access to the secret key is able to output a message μ and a signature σ for which $\mathsf{Verify}(\mu, \sigma, \mathsf{pk}) = 1$. A signature scheme is existentially unforgeable under an adaptive chosen message attack if the probability of an attacker producing a forgery is negligible, even if he has access to a signing oracle (in this case μ must not have been queried before to be considered a valid forgery). Again we refer to [KL07] for formal definitions, more refined security notions, and discussions.

3.2.2 Quantum Computing and Post-Quantum Cryptography

Most PKEs or DSSs that are commonly used in practice are either based on the factoring problem, like RSA, or the (elliptic curve) discrete logarithm problem (DLP), like DSA or ECDSA. However, since Shor's seminal result [Sho94] it is known that those asymmetric cryptosystems can be broken in polynomial time on a quantum computer (see [BBD08]). Currently, quantum computers are not available but in recent years there has been a big financial push, by both governments and private enterprises (see [Cha12, Koe13, RG13]), to construct a fully-functioning and large enough quantum computer. Such a device would have the capability to render the security guarantees of virtually all currently used public-key cryptography obsolete. An additional concern, given the dependence of basically all deployed schemes and protocols on only two very related problems, are further results in classical cryptanalysis. By demonstrating almost-polynomial time algorithms for the discrete logarithm problem in small-characteristic fields the authors of works like [Jou13, BGJT14] have also increased the awareness that cryptanalytic advances are possible even for relatively well researched problems. Note that symmetric encryption schemes like AES or hash algorithms are also affected by quantum computers [Gro96]. However, protection against attacks by quantum computers just requires the doubling of the symmetric key length and does thus not necessarily require new schemes and only moderately impacts performance.

However, when long-term security is required (see [BMV06]), the previously mentioned DLP or factoring-based public-key schemes are certainly risky to use, even today. The natural consequence is the need for more diversification and investigation of alternative asymmetric schemes based on problems that withstand classical cryptanalysis and resist attacks that rely on quantum computers. Such schemes are called *post-quantum* and the research area that deals with the design, analysis, and evaluation of cryptographic schemes that will still be secure in a world where quantum computers exist, is called *post-quantum cryptography* (PQC). In this field four major research areas have emerged, which are hash-based, code-based, multivariate-quadratic-equations-based, and lattice-based cryptography [BBD08]. Additionally, cryptosystems based on supersingular elliptic curve isogenies are also supposed to be secure against attacks by quantum computers [JF11].

Currently, most of the proposed schemes and underlying problems have not yet received as much attention as RSA or ECC and thus there is still need to establish confidence into the parameter selection approaches. Additionally, the key size of post-quantum schemes is usually larger than what is recommended for RSA or ECC and the operations carried out are different or more complicated so that the implementation on constrained devices can be challenging. For more details on PQC and more details on the previously mentioned alternative problems we refer to [BBD08].

3.3 Security Evaluation of Lattice-Based Cryptography

In this section we provide some references and information on the selection of parameters for ideal lattice-based cryptography and on the estimation of the security level provided by these parameter sets. However, cryptanalysis of lattice constructions is not in the scope of this thesis. We also refer to Section 3.4.2 for details on the parameter selection of the RLWEenc PKE, and to Section 3.5.2 and Section 3.6 for details on the parameter selection of GLP and BLISS signatures, respectively. Additionally, results of security reductions of LWE or RLWE to hard lattice problems and the asymptotic hardness of these underlying problems are given in Section 2.3.

In this thesis we are dealing with practical implementations so it is clear that concrete parameters are necessary and that an asymptotic security analysis is not sufficient. Additionally, the choice of parameters can have a big impact on the performance of an implementation. As we are interested in an evaluation of lattice-based cryptography in comparison to other post-quantum and classical proposals, it is important to compare schemes of similar security levels. As a consequence, the estimation of a bit-security level is necessary that relates the hardness of breaking the asymmetric scheme to the hardness of breaking a symmetric encryption scheme using brute force.

One approach for parameter selection is to rely on security reductions. In this case only the concrete hardness of the underlying lattice problems (e.g., SVP, CVP, or SIVP) has to be analyzed. However, most reductions are not tight so that this approach might result in schemes with parameters that are too large to be competitive. If the tightness is ignored this approach could also lead to average case problems that are reduced to easy instances of worst-case problems. As an example, Regev's worst-case to average-case reduction (see Theorem 3) provides a reduction from LWE to uSVP_γ and SIVP_γ of dimension n where $\gamma = \mathcal{O}(n/\alpha)$ but requires a quantum algorithm. The classical reductions [Pei09, BLP$^+$13] do not preserve the dimension and reduces a problem in dimension n to a problem in dimension $\sqrt{n}$.

A more direct method for parameter selection is the development of attacks on concrete average case problems like LWE or RLWE. In this case the security proof is used as an argument that shows that the scheme does not possess structural weaknesses and that possible attacks can be mitigated by raising the parameters, while retaining the constructions. However, bit security is estimated based on an extrapolation of the runtime of the best known attacks.

In this thesis we only use ideal lattices and the additional structure of ideal lattices might lead to more efficient attacks than for standard (or random) lattices. A general rule of thumb is that too much or unnecessary structure should be avoided. However, the benefits of ideal lattices seem to outweigh the risk of more efficient attacks - at least given the current state-of-the-art in cryptanalysis. The security argument for using ideal lattices is that current algorithms that attack LWE or SIS do not profit from the additional structure. As an example, some attacks require the reduction of the lattice using variants of the LLL algorithm [LLL82] and these algorithms do not retain the structure. As a consequence, problems like RLWE are currently analyzed by looking at the best algorithms to solve LWE. An open problem is to construct algorithms that target ideal lattices or RLWE and RSIS directly and that provide a meaningful improvement in runtime over the best generic attacks on LWE or SIS. Some ideas for an algorithm that could exploit the multiplicative structure of ideals were provided in a blog post by Bernstein [Ber14]. However, an actual implementation and analysis of the attack idea is still an open problem. Additionally, care has to be taken when selecting the ring used to instantiate an ideal lattice-based cryptosystem. The security reduction in [LPR10b] of RLWE holds only for some rings and in [ELOS15] Elias et al. show some weak instances.

A serious issue when using the best known attacks for parameter estimation is to identify which algorithms perform best. Due to the large number of parameters (e.g., σ, n, q) that impact the efficiency of certain algorithms it appears necessary to consider many approaches simultaneously to find the most efficient one. One approach to solve LWE directly is the Blum, Kalai and Wasserman (BKW) [BKW03] algorithm that has been designed to solve the LPN problem. In [Fit14, ACF+15] an analysis of the complexity of BKW, when used to solve LWE, is provided. However, the biggest obstacle for the application of BKW is the requirement of a subexponential number of samples, which are usually not available in practice.

One possible approach to obtain LWE parameters is an analysis relying on variants of Babai's nearest planes algorithm [Bab86]. The general idea is to convert an LWE instance into a CVP instance which is then solved using a CVP solver. The usage of the nearest planes algorithm for the LWE case has, for example, been evaluated by Lindner and Peikert [LP11] and Liu and Nguyen [LN13]. A practical evaluation of the runtime of nearest planes is given in [BBD+14]. To increase the success rate of the nearest planes algorithm a basis reduction algorithm like BKZ [CN11] can be applied. Another approach to solve LWE is the reduction of LWE to uSVP and the usage of an SVP solver. This so called "embedding approach" or "embedding attack" is analyzed in [AFG13]. However, still no attack has emerged that works best in all cases.

3.4 A RLWE-Based Public-Key Encryption Scheme (RLWEenc)

The properties of the RLWE problem [LPR10b] can be used to realize a semantically secure public-key encryption scheme with a reduction to decisional ring learning with errors (RdLWE). The general idea is to hide the secret key inside of a RLWE sample that becomes the public key and to mask the message with a RLWE sample. Thus the public key and each ciphertext appear uniformly random to a passive adversary and semantic security is achieved. The scheme we are going to discuss is usually attributed either to Lyubashevsky, Peikert, and Regev [LPR10b] or Lindner and Peikert [LP11]. The first peer-reviewed publication of the scheme can be found in the work by Lindner and Peikert [LP11] and then later on in the full version of the paper by Lyubashevsky, Peikert, and Regev [LPR12]. However, the presentation of [LPR10b] by Peikert at Eurocrypt'10 already contained the description of the scheme [LPR10c]. Due to the unclear origin we refer to the scheme as RLWEenc.

3.4.1 Definition

The RLWEenc PKE is instantiated in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$. Its key generation procedure $\mathsf{RLWEenc}_{\mathsf{gen}}$ is detailed in Algorithm 8 and just requires the random sampling of two noise polynomials $\mathbf{r}_1, \mathbf{r}_2$ from a discrete Gaussian distribution with Gaussian parameter σ. The

Algorithm 8 RLWEenc Key Generation

```
Precondition: Access to global constant $\mathbf{a}$ that was uniformly random chosen from $\mathcal{R}_q$
func $\mathsf{RLWEenc}_{\mathsf{gen}}()$
    $\mathbf{r}_1, \mathbf{r}_2 \xleftarrow{\$} D_{\mathbb{Z}^n,\sigma}$
    $\mathbf{p} \leftarrow \mathbf{r}_1 - \mathbf{a}\mathbf{r}_2$
    Return $(\mathsf{pk}, \mathsf{sk}) = (\mathbf{p}, \mathbf{r}_2)$
end func
```

Algorithm 9 RLWEenc Encryption

```
Precondition: Access to global constant $\mathbf{a}$
func $\mathsf{RLWEenc}_{\mathsf{enc}}(\mathsf{pk}{=}\mathbf{p}, \mu{\in}\{0,1\}^n)$
    $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \xleftarrow{\$} D_{\mathbb{Z}^n,\sigma}$
    $\bar{\mathbf{m}} \leftarrow \mathsf{Encode}(\mu)$
    $\mathbf{c}_1 \leftarrow \mathbf{a}\mathbf{e}_1 + \mathbf{e}_2$
    $\mathbf{c}_2 \leftarrow \mathbf{p}\mathbf{e}_1 + \mathbf{e}_3 + \bar{\mathbf{m}}$
    Return $c = [\mathbf{c}_1, \mathbf{c}_2]$
end func
```

Algorithm 10 RLWEenc Decryption

```
func $\mathsf{RLWEenc}_{\mathsf{dec}}(c = [\mathbf{c}_1, \mathbf{c}_2], \mathsf{sk} = \mathbf{r}_2)$
    Return $\mu \leftarrow \mathsf{Decode}(\mathbf{c}_1\mathbf{r}_2 + \mathbf{c}_2)$
end func
```

public key is $\mathbf{p} \leftarrow \mathbf{r}_1 - \mathbf{a}\mathbf{r}_2$. According to [LP11] the polynomial $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ can be chosen during key generation (as part of each public key) or regarded as a global constant. We use $\mathbf{a}$ as global constant as this allows further optimizations but note that care has to be taken that $\mathbf{a}$ is generated from a public verifiable random generator (e.g., by using a binary interpretation of π). The polynomial $\mathbf{r}_1$ is used only during key generation and discarded afterwards, while $\mathbf{r}_2$ is the secret key. Extraction of $\mathbf{r}_2$ from the public key $\mathbf{p}$ is equivalent to solving the RSLWE problem with one given sample.

The encryption procedure $\mathsf{RLWEenc}_{\mathsf{enc}}$, as given in Algorithm 9, requires the sampling of three noise polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ from the discrete Gaussian distribution $D_{\mathbb{Z}^n,\sigma}$. To hide the message in the ciphertext, it is encoded as $\bar{\mathbf{m}}$ and added to $\mathbf{p}\mathbf{e}_1 + \mathbf{e}_3$. The ciphertext $c \in \mathcal{R}_q \times \mathcal{R}_q$ consists of $\mathbf{c}_1$ and $\mathbf{c}_2$ which are basically both RLWE samples in Hermite normal form. Security against chosen plaintext attacks (CPA) follows from the fact that everything that is returned by the encryption algorithm is indistinguishable from random. We refer to [LP11, Theorem 3.2] for a security proof that assumes hardness of DLWE and which could also be applied to the ring version we are discussing (in this case hardness is based on RDLWE).

The decryption procedure $\mathsf{RLWEenc}_{\mathsf{dec}}$ is given in Algorithm 10. It requires knowledge of the secret key $\mathbf{r}_2$ since otherwise the large term $\mathbf{a}\mathbf{e}_1\mathbf{r}_2$ cannot be eliminated when computing $\mathbf{c}_1\mathbf{r}_2 + \mathbf{c}_2$. The encoding of the n-bit message μ is necessary as some small noise (i.e., $\mathbf{e} = \mathbf{e}_1\mathbf{r}_1 + \mathbf{e}_2\mathbf{r}_2 + \mathbf{e}_3$) is still present after calculating $\mathbf{c}_1\mathbf{r}_2 + \mathbf{c}_2$ and would prohibit the retrieval of the message after decryption. Note that the noise is relatively small as all noise terms are sampled from a narrow Gaussian distribution $D_{\mathbb{Z}^n,\sigma}$. To deal with the remaining noise, Lindner and Peikert [LP11] proposed threshold encoding functions for individual coefficients which we implicitly also apply to polynomials. For a coefficient of the message $\mu' \in \mathbb{Z}_q$ they define

$$\mathsf{Encode}(\mu') = \mu' \cdot \left\lfloor \frac{q}{2} \right\rfloor$$

and the decoding of $\bar{\mu}' \in \mathbb{Z}_q$ as

$$\mathsf{Decode}(\bar{\mu}') = \left\{ \begin{array}{l} \text{return 1 iff } \bar{\mu}' \in \left[- \left\lfloor \frac{q}{4} \right\rfloor, \left\lfloor \frac{q}{4} \right\rfloor\right) \subset \mathbb{Z}_q \\ \text{return 0 otherwise.} \end{array} \right\}$$

Thus the error tolerance is $t = \left\lfloor \frac{q}{4} \right\rfloor$ and decryption correctness is assured as long as $e_i \in [-t, t)$ for each coefficient e_i of $\mathbf{e}$. In other words, the maximum error added to each coefficient must not be larger than or equal to $\left\lfloor \frac{q}{4} \right\rfloor$ in order to decrypt correctly. The probability of a decryption error is mainly dominated by the tailcut and the standard deviation of the Gaussian parameter $\sigma = \frac{s}{\sqrt{2\pi}}$. Decreasing s decreases the error probability but also negatively affects the security of the scheme and a trade-off between correctness and security has to be found.

3.4.2 Parameter Selection

In Table 3.1 we summarize previously proposed RLWEenc parameter sets (n, q, s) for $s = \sigma\sqrt{2\pi}$ with regard to security as well as ciphertext, public key, and secret key size. Lindner and Peikert [LP11] were the first to provide parameter sets and chose $(192, 4093, 8.87)$, $(256, 4093, 8.35)$ and $(320, 4093, 8.00)$ which provide 62.8, 105.5, and 156.9 bits of security, respectively, according to a refined security analysis by Liu and Nguyen [LN13] for standard LWE. In the original work of Lindner and Peikert [LP11] the parameter sets were labeled low security, medium security, and high security, where medium security was considered equivalent to the security of the symmetric AES-128 block cipher. To support the NTT, Göttert et al. [GFS$^+$12] introduced hardware-friendly parameter sets for medium and for high security with $(256, 7681, 11.31)$ and $(512, 12289, 12.18)$. These sets are not explicitly analyzed in [LN13] but we expect the $(256, 7681, 11.31)$ parameter set to provide similar security to the $(256, 4093, 8.35)$ one. For the two parameter sets by Göttert et al., n is chosen as a power of two and q as a prime such that $q = 1 \bmod 2n$ and as a consequence, the NTT and negative wrapped convolution property can be used for efficient implementation. For the ciphertext expansion given in Table 3.1 we assume that the complete polynomial $\mathbf{c}_1, \mathbf{c}_2$ is transmitted (see Section 5.2.2 for improvements) and that the message size is n bits so that we get an expansion factor of $2\lceil \log_2(q) \rceil$ per message bit. We also set $\mathsf{pk} = (\mathbf{a}, \mathbf{p})$ but it would be possible to set $\mathbf{a}$ as a global constant as in Algorithm 8. The secret key size depends on the tailcut τ of the discrete Gaussian. For compliance with the security reduction the tailcut is set to $\tau = 13.4$ for 128 bits of security by most authors [DG14, BCNS15]. Thus coefficients of the secret key are in $[-\tau\sigma, \tau\sigma]$. With the help of a compression algorithm the secret key size could be further reduced (see Section 5.2.2) but this would also require resources for decoding the secret key during decryption.

Table 3.1: Security levels, ciphertext sizes, public key sizes, and secret key sizes of previously proposed RLWEenc parameter sets.

Set	Parameter (n, q, s)	Security bits	$\lvert c\rvert = \lvert\mathbf{c}_1, \mathbf{c}_2\rvert$ $2n\lceil\log_2(q)\rceil$ bits	$\lvert\mathsf{pk}\rvert = \lvert(\mathbf{a}, \mathbf{p})\rvert$ $2n\lceil\log_2(q)\rceil$ bits	$\lvert\mathsf{sk}\rvert = \lvert\mathbf{r}_2\rvert$ $n\lceil\log_2(\tau\sigma)\rceil$ bits
Ib	(192,4093,8.87) [LP11]	63	4,608	4,608	1,344
IIb	(256,4093,8.35) [LP11]	106	6,144	6,144	1,792
IIIb	(320,4093,8) [LP11]	157	7,680	7,680	2,240
Ia	(256,7681,11.32) [GFS+12]	106	6,656	6,656	1,792
IIa	(512,12289,12.18) [GFS+12]	256	14,336	14,336	3,584
Ic	(256,4096,8.35) [This work]	106	6,144	6,144	1,792

3.5 The GLP Signature Scheme

In [GLP12, GLP15] Güneysu, Lyubashevsky, and Pöppelmann proposed a signature scheme that is a combination of the schemes from [Lyu09] and [Lyu12] as well as an additional optimization that allows the reduction of the signature length by almost a factor of two. The security of the scheme is based on a particular version of the ring-SIS problem, where one is given an ordered pair of polynomials $(\mathbf{a}, \mathbf{t}) \in \mathcal{R}_q \times \mathcal{R}_q$ where $\mathbf{a}$ is chosen uniformly from $\mathcal{R}_q$ and $\mathbf{t} \leftarrow \mathbf{a}\mathbf{s}_1 + \mathbf{s}_2$, where $\mathbf{s}_1$ and $\mathbf{s}_2$ are chosen uniformly from $\{-k, \ldots, k\}^n$, and is asked to find an ordered pair $(\mathbf{s}_1', \mathbf{s}_2')$ such that $\mathbf{a}\mathbf{s}_1' + \mathbf{s}_2' = \mathbf{t}$. This can also be considered as the RDLWE problem with particularly "aggressive" parameters. The decisional compact knapsack ($\mathrm{DCK}_{q,n}$) problem is defined to be the problem of distinguishing between the uniform distribution over $\mathcal{R}_q \times \mathcal{R}_q$ and the distribution $(\mathbf{a}, \mathbf{a}\mathbf{s}_1 + \mathbf{s}_2)$ where $\mathbf{a}$ is uniformly random in $\mathcal{R}_q$ and $\mathbf{s}_1$ and $\mathbf{s}_2$ are uniformly random in $\{-1, 0, 1\}^n$. Currently, no efficient algorithms are known that can exploit that $\mathbf{s}_1$ and $\mathbf{s}_2$ are uniform (i.e., not Gaussian) and consist of only $-1/0/1$ coefficients. The Arora-Ge algorithm for solving LWE with small noise [AG11] does not apply to the DCK problem because this algorithm requires polynomially-many samples of the form $(\mathbf{a}_i, \mathbf{a}_i\mathbf{s} + \mathbf{e}_i)$, whereas in the DCK problem, only one such sample is given. Based on the lack of efficient algorithms it is conjectured in [GLP12, GLP15] that this problem is still hard. Recently, Micciancio and Peikert also showed that by imposing a limit on the number of samples, the LWE problem can still be hard with smaller noise [MP13].

3.5.1 Definition

In [GLP12, GLP15] two versions of the GLP signature scheme were proposed. In this section we just introduce the "optimized" variant which, compared to the "basic" variant, includes a compression algorithm for the second component of the signature.

The key generation algorithm $\mathsf{GLP}_{\mathsf{gen}}$ is described in Algorithm 11. Similar to RLWEenc a global constant $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ is used to reduce the size of the public key. Key generation basically requires sampling of random polynomials $\mathbf{s}_1, \mathbf{s}_2$ followed by one polynomial multiplication and an addition. The polynomials $\mathbf{s}_1$ and $\mathbf{s}_2$ have small coefficients in $\{-1, 0, 1\}$ while all coefficients of $\mathbf{a}$ and $\mathbf{t}$ are in the range $[-\frac{q-1}{2}, \frac{q-1}{2}]$. The private key sk consists of the values $\mathbf{s}_1, \mathbf{s}_2$ while $\mathbf{t}$

is the public key. Extraction of the secret key from the public key would require an adversary to solve the search version of the DCK problem.

Algorithm 11 GLP Key Generation

```
Precondition: Access to global constant $\mathbf{a}$
1: func $\mathsf{GLP}_{\mathsf{gen}}()$
2:     $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$
3:     $\mathbf{s}_1, \mathbf{s}_2 \xleftarrow{\$} \mathcal{R}_{q,1}$
4:     $\mathbf{t} \leftarrow \mathbf{a}\mathbf{s}_1 + \mathbf{s}_2$
5:     return $\mathsf{pk} = (\mathbf{t})$, $\mathsf{sk} = (\mathbf{s}_1, \mathbf{s}_2)$
6: end func
```

Algorithm 12 GLP Signing

```
Precondition: Access to global constant $\mathbf{a}$
1: func $\mathsf{GLP}_{\mathsf{sign}}(\mu \in \{0,1\}^*, \mathsf{sk}=(\mathbf{s}_1, \mathbf{s}_2))$
2:     $\mathbf{y}_1, \mathbf{y}_2 \xleftarrow{\$} \mathcal{R}_{q,k}$
3:     $\mathbf{c} \leftarrow \mathsf{H}\left((\mathbf{a}\mathbf{y}_1 + \mathbf{y}_2)^{(1)}, \mu\right)$
4:     $\mathbf{z}_1 \leftarrow \mathbf{s}_1\mathbf{c} + \mathbf{y}_1, \mathbf{z}_2 \leftarrow \mathbf{s}_2\mathbf{c} + \mathbf{y}_2$
5:     if $\mathbf{z}_1$ or $\mathbf{z}_2 \notin \mathcal{R}_{q,k-32}$, then goto step 1
6:     $\mathbf{z}_2' \leftarrow \mathsf{Compress}(\mathbf{a}\mathbf{z}_1 - \mathbf{t}\mathbf{c}, \mathbf{z}_2, p, k-32)$
7:     if $\mathbf{z}_2' = \perp$, then goto step 1
8:     return $(\mathbf{z}_1, \mathbf{z}_2', \mathbf{c})$
9: end func
```

Algorithm 13 GLP Verification

```
Precondition: Access to global constant $\mathbf{a}$
1: func $\mathsf{GLP}_{\mathsf{verify}}(\mu \in \{0,1\}^*, \mathsf{pk}=\mathbf{t}, (\mathbf{z}_1, \mathbf{z}_2', \mathbf{c}))$
2:     Accept iff $\mathbf{z}_1, \mathbf{z}_2' \in \mathcal{R}_{q,k-32}$ and
       $\mathbf{c} = \mathsf{H}\left((\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2' - \mathbf{t}\mathbf{c})^{(1)}, \mu\right)$
3: end func
```

The signing procedure $\mathsf{GLP}_{\mathsf{sign}}$ is detailed in Algorithm 12. In step 2, two polynomials $\mathbf{y}_1, \mathbf{y}_2$ are chosen uniformly at random with coefficients in the range $[-k, k]$. In step 3, a hash function H is applied on the higher-order bits of $\mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$ which outputs a polynomial $\mathbf{c}$ by interpreting the first 160-bit of the hash output as a sparse polynomial. In step 4, $\mathbf{y}_1$ and $\mathbf{y}_2$ are used to mask the private key by computing $\mathbf{z}_1$ and $\mathbf{z}_2$. The algorithm only continues if $\mathbf{z}_1$ and $\mathbf{z}_2$ are in the range $[-(k-32), k-32]$ and restarts otherwise. The polynomial $\mathbf{z}_2$ is then compressed into $\mathbf{z}_2'$ in step 6 by Compress. This compression is part of the aggressive size reduction of the signature $\sigma = (\mathbf{z}_1, \mathbf{z}_2', \mathbf{c})$ since only some portions of $\mathbf{z}_2$ are necessary to maintain the security of the scheme. For the implemented parameter set, Compress has a chance of failure of less than two percent which results in the restart of the whole signing process.

The verification algorithm $\mathsf{GLP}_{\mathsf{verify}}$ as described in Algorithm 13 first ensures that all coefficients of $\mathbf{z}_1, \mathbf{z}_2'$ are in the range $[-(k-32), k-32]$ and does not accept the invalid signature otherwise. In the next step, $\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2' - \mathbf{t}\mathbf{c}$ is computed, transformed into the higher-order bits (see below) and then hashed. If the polynomial $\mathbf{c}$ from the signature and the output of the hash match, the signature is valid and the algorithm accepts the signature.

The Compression Algorithm

In Algorithm 15 the transformation of a polynomial into a higher-order representation is described. This algorithm exploits the fact that every polynomial $\mathbf{y} \in \mathcal{R}_q$ can be written as

$$\mathbf{y} = \mathbf{y}^{(1)}(2(k-32)+1) + \mathbf{y}^{(0)}$$

where $\mathbf{y}^{(0)} \in \{-(k-32), (k-32)\}^n$. Due to this bijectional relationship, every polynomial $\mathbf{y}$ can be also written as the tuple $(\mathbf{y}^{(1)}, \mathbf{y}^{(0)})$ and we use the $\mathbf{y}^{(1)}$ notation to describe a transformation of $\mathbf{y} \in \mathcal{R}_q$ by HigherOrder (see Algorithm 14).

Algorithm 14 GLP Higher-Order Transformation $\mathbf{y}^{(1)}$

```
func HigherOrder($\mathbf{y} \in \mathcal{R}_q$, $k$)
  for i=0 to $n-1$ do
    $\mathbf{y}^{(0)}[i] \leftarrow \mathbf{y}[i] \mod (2(k-32)+1)$
    $\mathbf{y}^{(1)}[i] \leftarrow \frac{\mathbf{y}[i]-\mathbf{y}^{(0)}[i]}{2(k-32)+1}$
  end for
  return $y^{(1)}$
end func
```

Algorithm 15 describes the compression algorithm Compress which takes a polynomial $\mathbf{y}$, a polynomial $\mathbf{z}$ with small coefficients, and the security parameter k as well as q as input. It is designed to return a polynomial $\mathbf{z}'$ that is compact but still maintains the equality between the higher-order bits of $\mathbf{y}+\mathbf{z}$ and $\mathbf{y}+\mathbf{z}'$ so that $(\mathbf{y}+\mathbf{z})^{(1)} = (\mathbf{y}+\mathbf{z}')^{(1)}$. In particular, the parameters of the scheme are chosen in a way that the if-condition specified in step 4 is true only for rare cases. This is important since only values assigned to $\mathbf{z}'[i]$ in step 7 to step 16 can be efficiently encoded.

Instantiation of the Random Oracle

The hash function H maps an arbitrary-length message $\mu = \{1,0\}^*$ to a 512-coefficient polynomial with 32 coefficients in $\{-1,1\}$ and sets all other coefficients to zero. The whole process of generating this string and its transformation into a polynomial with the above described behavior is shown in Algorithm 16. In step 2 the message is concatenated with a binary representation of the polynomial $\mathbf{x}$ generated by the algorithm BinRep. It takes a polynomial $\mathbf{x} \in \mathcal{R}_q$ as input and outputs a (somehow standardized) binary representation of this polynomial. The 160-bit hash value is processed by partitioning it into 32 blocks of 5 side-by-side bits (beginning with the lowest ones) that each correspond to a particular region in the polynomial $\mathbf{c}$. These bits are $r_4r_3r_2r_1r_0$ where $(r_3r_2r_1r_0)_2$ represents the position in the region interpreted as a 4-bit unsigned integer and the bit r_4 determines if the value of the coefficient is -1 or 1.

3.5.2 Parameters and Security

In Table 3.2, two parameter sets, GLP-I and GLP-II, are provided. For some intuition on how these parameters were selected, how the security level has been computed, and a security proof in the random-oracle model we refer again to [GLP12, GLP15].

Algorithm 15 GLP Signature Compression

```
func Compress($\mathbf{y}, \mathbf{z}, p, k$)
  $uncompressed \leftarrow 0$
  for i=1 to n do
    if $|\mathbf{y}[i]| > \frac{p-1}{2} - k$ then
      $\mathbf{z}'[i] \leftarrow \mathbf{z}[i]$
      $uncompressed \leftarrow uncompressed + 1$
    else
      write $\mathbf{y}[i] = \mathbf{y}[i]^{(1)}(2k+1) + \mathbf{y}[i]^{(0)}$ where $-k \leq \mathbf{y}[i]^{(0)} \leq k$
      if $\mathbf{y}[i]^{(0)} + \mathbf{z}[i] > k$ then
        $\mathbf{z}'[i] \leftarrow k$
      else if $\mathbf{y}[i]^{(0)} + \mathbf{z}[i] < -k$ then
        $\mathbf{z}'[i] \leftarrow -k$
      else
        $\mathbf{z}'[i] \leftarrow 0$
      end if
    end if
  end for
  if $uncompressed \leq \frac{6kn}{p}$ then return $\mathbf{z}'$
  else return $\perp$
  end if
end func
```

Algorithm 16 GLP Random Oracle Instantiation

Precondition: Definition of a standard cryptographic hash function Hash that outputs a 160-bit string.

```
func H($\mathbf{x} \in \mathcal{R}_q, \mu \in \{0,1\}^*$)
  $r \leftarrow \mathsf{Hash}(\mu || \mathsf{BinRep}(\mathbf{x}))$
  for i=0 to $n-1$ do
    $\mathbf{c}[i] \leftarrow 0$
  end for
  for i=0 to 31 do
    $pos \leftarrow 8 \cdot r_{5i+3} + 4 \cdot r_{5i+2} + 2 \cdot r_{5i+1} + r_{5i}$
    if $r_{5i+4} = 0$ then
      $\mathbf{c}[i \cdot 16 + pos] \leftarrow -1$
    else
      $\mathbf{c}[i \cdot 16 + pos] \leftarrow 1$
    end if
  end for
end func
```

Table 3.2: GLP signature parameters [GLP12, GLP15].

Name of the scheme	GLP-I	GLP-II
Security	80 bits	256 bits
n	512	1024
p	8383489	16760833
k	2^{14}	2^{15}
Signature size	8,950 bit	18,800 bit
Secret key size	1,620 bit	3,250 bit
Public key size	11,800 bit	25,000 bit
Repetition rate	7	7
Root Hermite factor	1.0066	1.0035

In general, the security of the signature scheme is based on the $\mathrm{DCK}_{q,n}$ problem and the hardness of finding a preimage in the hash function. As the signature scheme is based on the Fiat-Shamir transform [FS86] one only needs random oracles that output λ bits (i.e., collision-resistance is not a requirement) to achieve λ bits of security. While finding collisions in the random oracle does allow the *valid* signer to produce two distinct messages that have the same signature, this does not constitute a break. Finding a preimage in the hash function has classical time complexity of 2^l but is lowered to $2^{l/2}$ by Grover's quantum algorithm [Gro96]. As $l = 160$ output bits from the hash function are used, the implemented scheme is supposed to achieve a security level of roughly 80 bits of security against attacks by a quantum computer on the hash function.

In [GLP12] the security level of parameter set GLP-I is estimated to be 100 bits of security based on the hardness of solving the underlying lattice problem. However, in [GLP15] the claimed security level is reduced to 80 bits due to experiments given in the full version of the BLISS signature paper [DDLL13a]. The general approach for security estimation in [GLP12, GLP15] is based on the so-called root Hermite factor. Gama and Nguyen [GN08b] and Chen and Nguyen [CN11] carried out various experiments and state that a root Hermite factor of 1.01 is achievable now, a factor of 1.007 seems to have around 80 bits of security, and a factor of 1.005 has more than 256-bit security. Thus GLP-I with root Hermite factor 1.0066 is somewhere around 80 bits of security, while the second parameter set GLP-II is supposed to provide more than 256 bits of security. However, it does not seem clear whether these attacks could be accelerated using a quantum computer.

3.6 The Bimodal Lattice-Based Signature Schemes (BLISS)

The most efficient instantiation of the BLISS signature scheme [DDLL13a] is based on ideal lattices and operates on polynomials over the ring $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle \mathbf{x}^n + 1\rangle$. As the design is similar to GLP and also uses the Fiat-Shamir transformation to turn an identification scheme into a signature scheme we provide a more compact description. Note that BLISS also requires the

Algorithm 17 BLISS Key Generation

```
func $\mathsf{BLISS}_{\mathsf{gen}}()$
    Choose $\mathbf{f}, \mathbf{g}$ as uniform polynomials with exactly $d_1 = \lceil \delta_1 n \rceil$ entries in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ entries in $\{\pm 2\}$
    $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g} + 1)^t$
    if $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5 \cdot (\lceil \delta_1 n \rceil + 4 \lceil \delta_2 n \rceil) \cdot \kappa$ then restart
    $\mathbf{a}_q \leftarrow (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (restart if $\mathbf{f}$ is not invertible)
    return $(pk = \mathbf{A}, sk = \mathbf{S})$ where $\mathbf{A} = (\mathbf{a}_1 = 2\mathbf{a}_q, q - 2) \bmod 2q$
end func
```

Algorithm 18 BLISS Signing

```
func $\mathsf{BLISS}_{\mathsf{sign}}(\mu \in \{0,1\}^*, \mathsf{pk}{=}\mathbf{A}, \mathsf{sk}{=}\mathbf{S})$
    $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n, \sigma}$
    $\mathbf{u} \leftarrow \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
    $\mathbf{c} \leftarrow H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
    Choose a random bit $b$
    $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$
    $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \mathbf{c}$
    Continue with probability $1 \Big/ \left( M \exp\left( -\frac{\|\mathbf{Sc}\|^2}{2\sigma^2} \right) \cosh\left( \frac{\langle \mathbf{z}, \mathbf{Sc} \rangle}{\sigma^2} \right) \right)$ otherwise restart
    $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
    return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
end func
```

Algorithm 19 BLISS Verification

```
func $\mathsf{BLISS}_{\mathsf{verify}}(\mu \in \{0,1\}^*, \mathsf{pk}{=}\mathbf{A}, \mathsf{sk}{=}\mathbf{S})$
    if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ then Reject
    if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ then Reject
    Accept iff $\mathbf{c} = H\big(\lfloor \zeta \mathbf{a}_1 \mathbf{z}_1 + \zeta q \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu\big)$
end func
```

usage of rejection sampling to ensure that the signature is independent of the secret key and distributed according to a discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$. The usage of bimodal Gaussian noise instead of uniform noise to hide the secret key is the biggest difference compared to GLP. This allows smaller signatures and much less rejections.

The key generation algorithm $\mathsf{BLISS}_{\mathsf{gen}}$ is provided in Algorithm 17 and is similar to the key generation algorithm of NTRU. First polynomials $\mathbf{f}$ and $\mathbf{g}$ with densities δ_1 and δ_2 are sampled such that they have $d_1 = \lceil \delta_1 n \rceil$ coefficients in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ coefficients in $\{\pm 2\}$. The secret key is computed as $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g} + 1)^t$ and the public key is $\mathbf{A} = (2\mathbf{a}_q, q - 2) \leftarrow \left(\frac{2 \cdot (2\mathbf{g}+1)}{\mathbf{f}}, q - 2 \right)$. For successful key generation, it is necessary to find a polynomial $\mathbf{f}$ that is invertible. If this is not the case, key generation is restarted. The size of the signature depends on the maximum possible norm of the vector $\mathbf{Sc}$, which is defined as $N_\kappa(\mathbf{S})$ and computed as

$$N_\kappa(\mathbf{S}) = \max_{\substack{I \subset \{1,\ldots,n\} \\ \#I = \kappa}} \sum_{i \in I} \left(\max_{\substack{J \subset \{1,\ldots,n\} \\ \#J = \kappa}} \sum_{j \in J} T_{i,j} \right),$$

where $\mathbf{T} = \mathbf{S}^t \cdot \mathbf{S} \in \mathbb{R}^{n \times n}$. A private key $\mathbf{S}$ whose $N_\kappa(\mathbf{S})$ value is too big is rejected, to keep the signature size small.

The signing algorithm $\mathsf{BLISS_{sign}}$ is described in Algorithm 18. First two masking polynomials $\mathbf{y}_1, \mathbf{y}_2$ are sampled where each coefficient is randomly chosen according to the Gaussian distribution D_σ. Then the polynomial $\mathbf{u}$ is computed as $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$ where multiplication by $\zeta = \frac{1}{q-2} \bmod 2q$ and reduction modulo $2q$ are necessary in order to achieve a low rejection rate. Note that despite the even modulus $2q$, it is still possible to compute $\mathbf{a}_1 \cdot \mathbf{y}_1$ using FFT or NTT-techniques that require a prime modulus for maximum efficiency. Then only the higher-order bits[1] $\lfloor \mathbf{u} \rceil_d$ of the polynomial $\mathbf{u}$ are hashed together with the message μ. This is done using a standard cryptographic hash function to instantiate the random oracle. The pseudo-random string returned by the hash function is then used to construct the sparse polynomial $\mathbf{c}$ with κ coefficients equal to one and the remaining coefficients set to zero. The secret key is multiplied by $\mathbf{c}$ (which depends on the message and the random $\mathbf{y}_1, \mathbf{y}_2$) and masked using the Gaussian distributed masking polynomials $\mathbf{y}_1, \mathbf{y}_2$. The rejection step is performed in step 8 to make the signature independent of the secret key. Due to this rejection step the signature generation restarts with a certain probability. Finally the size of the signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ is reduced by compressing $\mathbf{z}_2$ into $\mathbf{z}_2^\dagger$. To avoid explicit computation of $1/\cosh$ for the final rejection step, the authors of [DDLL13a] suggested that $\mathcal{B}_{1/\cosh(X)} = \mathcal{B}_{\exp(-|X|)} \oslash \left(\mathcal{B}_{1/2} \vee \mathcal{B}_{\exp(-|X|)}\right)$ and that this approach requires at most three calls to $\mathcal{B}_{\exp(-|X|)}$ on average.

The verification procedure $\mathsf{BLISS_{verify}}$ is given in Algorithm 19. To verify a signature it is checked if $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ is a valid signature of the message μ. First the l_2 and l_∞ norms of $\mathbf{z}_1$ and $\mathbf{z}_2$ are computed and the signature is rejected if they are too large (depending on the parameters B_2 and B_∞). The actual verification is done by computing $H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$ and testing whether the result is equal to $\mathbf{c}$. Parameters as well as achievable signature and key sizes are listed in Table 3.3. The parameters are based on an extensive security analysis provided in [DDLL13b].

3.7 The Somewhat Homomorphic Encryption Scheme YASHE

The homomorphic encryption scheme YASHE [BLLN13b] is based on the multi-key fully homomorphic encryption scheme from [LTV12] and the modified, provably-secure version of NTRU presented in [SS11]. Stehlè and Steinfeld show in [SS11] how to modify the NTRU public-key scheme [HPS98] in order to make it provably secure based on the RLWE problem [LPR10b]. López-Alt et al. [LTV12] demonstrate that this modified NTRU encryption can be used to construct a multi-key fully homomorphic encryption scheme. However, they only prove security under an additional assumption, the so-called decisional small polynomial ratio (DSPR) assumption. In [BLLN13b], it is shown that a scale-invariant version of the modified NTRU scheme can be used as a basis for a fully homomorphic scheme with security only relying on the assumption that the RLWE problem is hard. The homomorphic multiplication operation in this scheme is very costly and [BLLN13b] contains a more practical variant, denoted as YASHE', with a much more efficient homomorphic multiplication algorithm. This is achieved by reintroducing the DSPR assumption in addition to RLWE. YASHE' is leveled homomorphic and

---

[1] For any integer x, the d high-order bits of x are denoted by $\lfloor x \rceil_d$ so that x can be written as $x = \lfloor x \rceil_d \cdot 2^d + [x \bmod 2^d]$.

Table 3.3: BLISS signature parameters [DDLL13a].

Name of the scheme	BLISS-I	BLISS-II	BLISS-III	BLISS-IV
Security	128 bits	128 bits	160 bits	192 bits
Optimized for	Speed	Size	Security	Security
(n, q)	(512,12289)	(512,12289)	(512,12289)	(512,12289)
Lattice Dim. $m = 2n$	1024	1024	1024	1024
Secret key densities δ_1, δ_2	0.3 , 0	0.3 , 0	0.42 , 0.03	0.45, 0.06
Gaussian std. dev. σ	215.73	107.86	250.54	271.93
Max Shift/std. dev. ratio α	1	.5	.7	.55
Weight of the challenge κ	23	23	30	39
Secret key N_κ-Threshold C	1.62	1.62	1.75	1.88
Dropped bits d in $\mathbf{z}_2$	10	10	9	8
Verif. thresholds B_2, B_∞	12872, 2100	11074, 1563	10206,1760	9901, 1613
Repetition rate	1.6	7.4	2.8	5.2
Entropy of challenge $\mathbf{c} \in \mathbb{B}^n_\kappa$	132 bits	132 bits	161 bits	195 bits
Signature size	5,600 bit	5,000 bit	6,000 bit	6,500 bit
Secret key size	2,000 bit	2,000 bit	3,000 bit	3,000 bit
Public key size	7,000 bit	7,000 bit	7,000 bit	7,000 bit
Root Hermite factor	1.0037	1.0037	1.0033	1.0031

due to its scale-invariance achieves its homomorphic capabilities without the modulus switching technique. Next, we describe this scheme, which we use under the name YASHE in this thesis.

The system parameters are fixed as follows: a positive integer $m = 2^k$ that determines the ring $R = \mathbb{Z}[\mathbf{x}]/(\mathbf{x}^n + 1)$ and its dimension $n = \varphi(m) = m/2$, two moduli q and t with $1 < t < q$, discrete probability distributions $\chi_{\text{key}}, \chi_{\text{err}}$ on R, and an integer base $w > 1$. We view R to be the ring of polynomials with integer coefficients taken modulo the m-th cyclotomic polynomial $\mathbf{x}^n + 1$. Let $R_q = R/qR \cong \mathbb{Z}_q[\mathbf{x}]/(\mathbf{x}^n + 1)$ be defined by reducing the elements in R modulo q, similarly we define R_t.

A polynomial $\mathbf{a} \in R_q$ can be decomposed using base w as $\mathbf{a} = \sum_{i=0}^{\ell_{w,q}-1} \mathbf{a}_i w^i$, where the $\mathbf{a}_i \in R$ have coefficients in $(-w/2, w/2]$. The scheme YASHE makes use of the functions $\mathsf{Dec}_{w,q}(\mathbf{a}) = ([\mathbf{a}_i]_w)_{i=0}^{\ell_{w,q}-1}$ and $\mathsf{Pow}_{w,q}(\mathbf{a}) = ([\mathbf{a}w^i]_q)_{i=0}^{\ell_{w,q}-1}$, where $\ell_{w,q} = \lfloor \log_w(q) \rfloor + 1$. Both functions take a polynomial and map it to a vector of polynomials in $R^{\ell_{w,q}}$. They satisfy the scalar product property $\langle \mathsf{Dec}_{w,q}(\mathbf{a}), \mathsf{Pow}_{w,q}(\mathbf{b}) \rangle = \mathbf{ab} \pmod q$.

The homomorphic encryption scheme YASHE consists of algorithms for key generation, encryption and decryption. The evaluation functions that allow computation on encrypted data are the two functions Add and Mult. The latter consists of two parts, the rounded multiplication RMult and the key switching step KeySwitch. The scheme is defined as follows:

$\mathsf{Gen}(d, q, t, \chi_{\text{key}}, \chi_{\text{err}}, w)$: Sample $\mathbf{f}' \leftarrow \chi_{\text{key}}$ until $\mathbf{f} = [t\mathbf{f}' + 1]_q$ is invertible modulo q. Compute the inverse $\mathbf{f}^{-1} \in R$ of $\mathbf{f}$ modulo q, sample $\mathbf{g} \leftarrow \chi_{\text{key}}$ and set $\mathbf{h} = [t\mathbf{g}\mathbf{f}^{-1}]_q$. Sample $\mathbf{e}, \mathbf{s} \leftarrow \chi_{err}^{\ell_{w,q}}$, compute $\gamma = [\mathsf{Pow}_{w,q}(\mathbf{f}) + \mathbf{e} + \mathbf{h} \cdot \mathbf{s}]_q \in R^{\ell_{w,q}}$ and output $(\mathsf{pk}, \mathsf{sk}, \mathsf{evk}) = (\mathbf{h}, \mathbf{f}, \gamma)$.

Table 3.4: YASHE parameter sets and supported number of multiplicative levels for different plaintext moduli t.

Set	n	q	q'	$\ell_{w,q}$	Levels			
					$t = 2^{20}$	$t = 2^{10}$	$t = 2^{5}$	$t = 2$
I	4096	$2^{124} - 2^{64} + 1$	$2^{262} - 2^{56} + 1$	2	0	1	1	1
II	16384	$2^{512} - 2^{32} + 1$	$2^{1040} - 2^{32} + 1$	8	6	9	11	14

Following the analysis in [LN14], these parameters provide at least 80 bits of security. The parameter for sampling the discrete Gaussian error distribution is $s = 8$.

$\mathsf{Encrypt}(\mathbf{h}, \mathbf{m})$: For a message $\mathbf{m} \in R/tR$, sample $\mathbf{s}, \mathbf{e} \leftarrow \chi_{\text{err}}$, scale $[\mathbf{m}]_t$ by the value $\lfloor q/t \rfloor$, and output $\mathbf{c} = \left[\left\lfloor \frac{q}{t} \right\rfloor [\mathbf{m}]_t + \mathbf{e} + \mathbf{hs}\right]_q \in R$.

$\mathsf{Decrypt}(\mathbf{f}, \mathbf{c})$: Decrypt $\mathbf{c}$ as follows. First compute the ring product $[\mathbf{fc}]_q$ modulo q, scale it down by the factor t/q over the rational numbers, round it and reduce it modulo t, i.e., output $\mathbf{m} = \left[\left\lfloor \frac{t}{q} [\mathbf{fc}]_q \right\rceil\right]_t \in R$.

$\mathsf{Add}(\mathbf{c}_1, \mathbf{c}_2)$: Add the two ciphertexts modulo q, i.e., output $\mathbf{c}_{\text{add}} = [\mathbf{c}_1 + \mathbf{c}_2]_q$.

$\mathsf{RMult}(\mathbf{c}_1, \mathbf{c}_2)$: Compute the product $\mathbf{c}_1\mathbf{c}_2$ without reduction modulo q over the integers, scale by t/q, round the result and reduce modulo q to output

$$\tilde{\mathbf{c}}_{\text{mult}} = \left[\left\lfloor \frac{t}{q} \mathbf{c}_1 \mathbf{c}_2 \right\rceil\right]_q .$$

$\mathsf{KeySwitch}(\tilde{\mathbf{c}}_{\text{mult}}, \mathsf{evk})$: Compute the w-decomposition vector of $\tilde{\mathbf{c}}_{\text{mult}}$ and output the scalar product with the evaluation key evk and reduce modulo q: $\mathbf{c}_{\text{mult}} = [\langle \mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}), \mathsf{evk} \rangle]_q$.

$\mathsf{Mult}(\mathbf{c}_1, \mathbf{c}_2, \mathsf{evk})$: First apply RMult to $\mathbf{c}_1$ and $\mathbf{c}_2$ and then $\mathsf{KeySwitch}$ to the result. Output the ciphertext $\mathbf{c}_{\text{mult}} = \mathsf{KeySwitch}(\mathsf{RMult}(\mathbf{c}_1, \mathbf{c}_2), \mathsf{evk})$.

In Table 3.4, we provide the implemented parameter sets and their number of supported multiplicative levels determined by the worst case bounds given in [BLLN13b]. The plaintext modulus in our implementation is $t = 1024$ for both parameter sets. Since changing t is relatively easy, we also give the number of multiplicative levels for various other choices to illustrate the dependence on t and possible trade-offs. The primes q and q' we use in our implementation are Solinas primes of the form $q = 2^y - 2^z + 1$, $y > z$ such that $q \equiv 1 \pmod{2n}$. In order to find a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$ that is needed for the NTT, we simply chose random non-zero elements in $a \in \mathbb{Z}_q$, until $a^{(q-1)/2n} \neq 1$ and $a^{(q-1)/2} = -1$ and then set $\psi = a^{(q-1)/2n}$.

According to the analysis in [LN14], the chosen moduli stay below the maximal bound to achieve 80 bits of security against the distinguishing attack with advantage 2^{-80} as discussed there. The error distribution χ_{err} is the n-dimensional discrete Gaussian with parameter $s = 8$ and the key distribution samples polynomials with uniform random values in $\{-1, 0, 1\}$.

Note that one ciphertext requires $n\lceil\log_2(q)\rceil$ bits (1 MiB for Set II) and the evaluation key is $(\ell_{w,q})n\lceil\log_2(q)\rceil$ bits large (8 MiB for parameter Set II).

The advantage of a scale-invariant scheme like YASHE is that modulus switching can be avoided so that a hardware implementation does not have to deal with various moduli. Additionally, YASHE supports one-element ciphertexts compared to Fan and Vercauteren's (FV) scale-invariant version of BGV [BGV12]. A detailed comparison of FV [FV12] with YASHE [BLLN13b], as well as details on the implementation and parameter selection can be found in [LN14].

Chapter 4

Polynomial Multiplication on Reconfigurable Hardware

In this chapter we introduce a hardware architecture for efficient polynomial multiplication using the number theoretic transform (NTT). Our implementation targets common parameters of lattice-based public-key encryption, signatures, and homomorphic encryption. Moreover, we introduce an efficient microcode engine that is built on top of the multiplier. It supports a simple instruction set suitable to realize the polynomial arithmetic of RLWEenc *encryption (see Chapter 5) as well as parts of* GLP *and* BLISS *signatures (see Chapter 7). This chapter is mainly based on work published in [PG12]. However, more detailed coverage of efficient methods to realize the processing element (PE) and general improvements to the architecture have been added. Details on the microcode engine were published in [PG13] and the instruction set appeared in [GLP15].*

Contents of this Chapter

4.1 Introduction

When considering the performance of proposed ideal lattice-based cryptosystems like RLWEenc [LPR10b, LP11], GLP [GLP12], and BLISS [DDLL13a], the most common and usually also most expensive operation is polynomial multiplication in $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$[1]. This is also the reason for the often claimed asymptotic speed advantage of ideal lattice-based schemes over number theoretical constructions, as polynomial multiplication in $\mathcal{R}_q$ is equivalent to the

---

[1] In this context polynomial multiplication is comparable to modular exponentiation being the basic operation required for RSA and point multiplication for elliptic curve cryptography (ECC).

negacyclic or negative-wrapped convolution product, which can be efficiently computed with $\mathcal{O}(n \log n)$ multiplications in $\mathbb{Z}_q$ using the NTT (see Section 2.4.2).

At the time this research was carried out[2], not much was known about the actual speed of implementations of polynomial multiplication targeting ideal lattice-based cryptography. The core by Györf et al. [GCHB12] was designed only for small parameter sets and the implementation of RLWEenc by Göttert et al. [GFS⁺12] consumed lots of FPGA resources to realize a parallel NTT polynomial multiplier.

To enable the evaluation of a whole range of schemes on reconfigurable hardware (e.g., RLWEenc, GLP, and homomorphic encryption with small parameters [NLV11]) a more flexible and also resource efficient multiplier was required. But as polynomial multiplication is not the only operation, also addition and subtraction of polynomials, storage of key or temporary polynomials, as well as sampling of random polynomials from certain distributions (e.g., uniform or Gaussian) should be supported. As a consequence, a microcode approach seems advantageous in which a state machine can issue commands that are decoded and then executed by a processor-like design.

4.1.1 Related Work

Techniques for the efficient implementation of the FFT in hardware have been explored in numerous works like [Pea68, Ber69, WLT07, GRH11] but the focus is usually the computation of approximate results due to the required complex floating-point arithmetic. We are not aware of any works published before the year 2012 that specifically deal with polynomial multiplication targeting ideal lattice-based cryptography, e.g., encryption [LPR10b] or signature schemes [Lyu12, GLP12]. First works were [GCHB12, GFS⁺12] and since then the state-of-the-art regarding the efficient hardware implementation of polynomial multiplication for ideal lattice-based cryptography has been continuously improved. An optimized polynomial multiplier was proposed by Aysu, Patterson and Schaumont [APS13]. A microcode engine with an even more efficient multiplier was introduced by Roy et al. [RVM⁺14] and used to implement RLWEenc. An implementation targeting larger parameter sets required for somewhat homomorphic encryption (SHE) was proposed in [CMV⁺14]. Examples of previous implementations of polynomial multiplication targeting Galois fields $\mathrm{GF}(2^n)$ used in elliptic curve cryptography (ECC) are [vzGS05, BS06].

4.1.2 Contribution

In this section we present a flexible and extensible FPGA implementation of polynomial multiplication based on the NTT. All arithmetic operations are specifically optimized for the ring $\mathcal{R}_q = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$, which is heavily used in ideal lattice-based cryptography. We further show how to efficiently realize basic arithmetic and the required NTT butterfly for specific moduli q. Building on top of these results our implementation supports a broad range of parameters and we give instantiations for previously proposed parameter sets for public-key encryption and SHE. Our design is scalable, has small area consumption on a low-cost FPGA, and offers decent performance compared to general purpose computers. Additionally, we present a microcode engine

---

[2] The first publication in which our multiplier appeared is [PG12].

with the polynomial multiplier as core component that allows polynomial additions and subtractions, storage of polynomials in registers, and random sampling of polynomials. This processor has been used as a building block for the implementations covered in Chapter 5 (RLWEenc) as well as Chapter 7 (GLP and BLISS).

4.2 Design Decisions

In this section we deal with the overall design of a polynomial multiplier using the NTT for the implementation of lattice-based cryptography. The necessary mathematical background is covered in Section 2.4.2. When we write polynomial multiplication as $\mathbf{c} = \mathsf{INTT}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ for $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{R}_q$ this usually also includes multiplication by powers of ψ and ψ^{-1}.

4.2.1 Previous Work Unrelated to Lattice-Based Cryptography

The implementation of the FFT in hardware has been extensively researched for more than forty years and a large variety of approaches exist for certain demands regarding throughput, power consumption as well as memory and resource usage, or latency [Pea68, Ber69]. A common approach is a pipelined implementation [DMH+11] where usually one stage of the FFT is computed in one clock cycle in parallel. However, while being extremely fast, this approach is expensive in terms of hardware resources and the FFT can also be implemented memory-based [WLT07] with the coefficients and constants being stored in one large memory block. The processing is usually controlled by a digital address generator and performed iteratively by only one or a small number of processing elements (PE) carrying out the actual computations. For radix-2 algorithms the PE usually resembles the well-known butterfly structure and is used to multiply elements of the processed vector by powers of ω (twiddle factors). In some cases the length of the processed sequence is fixed but there exist also very flexible, variable length, and memory-based FFT implementations [GRH11]. A more detailed characterization of FFT architectures can be found in [SSHA08]. The implementation of the NTT on graphic cards is described in [Eme09]. A cryptographic processor for performing elliptic curve cryptography (ECC) in the frequency domain is presented in [BKPS07].

4.2.2 Design Decisions for Lattice-Based Cryptography

As lattice-based cryptography is a very active field of research and as the hardness of underlying problems is not fully understood, no standard parameters like specific moduli or certain numbers of polynomial coefficients have emerged yet. This is different for established cryptosystems, e.g., elliptic curve cryptography (ECC). For the ECC example, NIST primes have been specified for efficient modular reduction [GP08] and just a few parameter sets have to be considered which allows much more optimized implementations. As such standardization is not yet achieved for lattice-based cryptography, we have made our implementation of polynomial multiplication as generic as possible to support a large number of application scenarios – consequently this also gives room for parameter or application specific tuning of the architecture (see [APS13, RVM+14, CMV+14]). The only general requirement on supported parameters, which is also common in the literature, is that $q \bmod 2n \equiv 1$, n is a power of two, and q is a prime so that the NTT can be efficiently computed in $\mathbb{Z}_q$ (see parameters of schemes discussed in Chapter 3).

Moreover, a memory-based implementation where two polynomials $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$ are stored in two separate block memories that can be accessed by the NTT engine seems to offer a good trade-off between performance and area utilization. This allows to load $\mathbf{a}$ and $\mathbf{b}$ for multiplication into the registers, to perform the NTT in-place, and to overwrite one polynomial with the result $\mathbf{c} = \mathsf{INTT}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ – leaving the other in NTT form for further use. In this case the NTT engine is involved in two forward and one inverse transformation and is fully utilized the whole time.

Note also that the NTT is not considered as a very efficient method for most applications in signal processing as approximate solutions are usually sufficient (see [Per03] for calculations of the required accuracy for polynomial multiplication). Especially, the complexity of the butterfly structure in the PE requires a lot of resources as multiplication by the twiddle factor is just a general purpose multiplication followed by a modular reduction. When using the Fermat [AB74] or Mersenne [Rad72] number theoretic transform the butterfly can be implemented by shifters (as ω can be 2), no read-only memory (ROM) for twiddle factors is needed and the modular reduction is also heavily simplified [Bla10, Chap. 10] [BS06]. However, in this case the transform length has to be doubled (Theorem 5), more storage space for coefficients is needed, and the reduction $\bmod \langle \mathbf{x}^n + 1 \rangle$ has to be performed separately. But the most important observation when reusing the PE for polynomial multiplication is that the component-wise multiplication step $\mathbf{a} \circ \mathbf{b}$ requires a general purpose multiplication and cannot be implemented just with shifters. As our iterative implementation of the NTT engine with just one PE seems sufficiently fast for a lot of applications we do not implement parallel PEs and thus decided to reuse the multiplication hardware that we need for the component-wise multiplication step also in the NTT. This makes sense, as such hardware will be instantiated and would idle most of the time otherwise.

When utilizing (2) of Theorem 6 it seems that more arithmetic operations and additional ROM storage space are necessary due to the needed table entries for powers of ψ and ψ^{-1}. However, storage costs do not increase compared to a general purpose NTT with zero-padding (see Theorem 5). When multiplying two length n polynomials using a positive wrapped convolution we would have to append n zeros and thus storage of $2n$ twiddle factors with $2 \cdot \frac{n}{2}$ entries for the forward and $2 \cdot \frac{n}{2}$ entries for the inverse transform would be required[3]. When directly implementing the negative wrapped convolution we can use a transform size of n. In this case we need additional storage for the ψ^i values. But as $(\psi^2)^i = \omega^i$ we just have to store n powers of ψ to also store ω^0 to $\omega^{\frac{n}{2}-1}$ (the same hold for ω^{-1}). As a consequence, when implementing the negative wrapped convolution directly also $2n$ entries in the ROM are necessary.

4.3 Design of an Efficient NTT-Based Polynomial Multiplier

In this section we describe our FPGA implementation of a flexible NTT-based polynomial multiplication core specifically designed for high-performance lattice-based cryptography. For a detailed description and background on the NTT we refer to Section 2.4. The implementation is based on Algorithm 1.

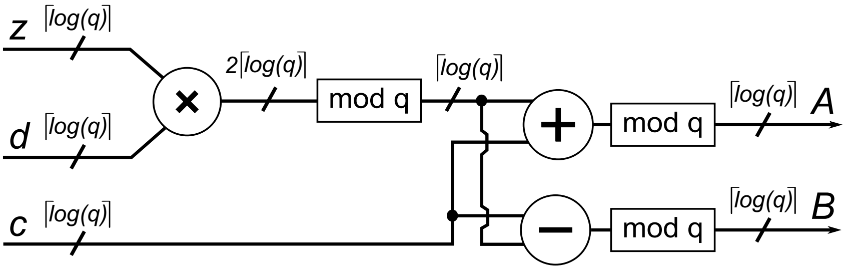

Figure 4.1: Block structure of the processing element (PE).

4.3.1 Processing Element

The processing element (PE) of the polynomial multiplier is depicted in Figure 4.1. It resembles the common butterfly structure of signal processing FFTs but relies on integer arithmetic and modulo reductions instead of floating or fixed point arithmetic. The PE is used to compute $A = c + zd \mod q$ and $B = c - zd \mod q$, where z is a power of ω, ψ, or their inverses. Thus one node of the NTT flow is computed every clock cycle (see step 14 and step 15 of Algorithm 1). In order to allow generic usage, a multiplier and a modular reduction circuit are synthesized for every given modulus $q < 2^{64}$. However, our implementation also supports plug-ins (using the VHDL `if generate` statement) that exploit specific structures in the prime number for a more efficient reduction, e.g., for the Fermat prime $2^{16} + 1$. At synthesis time, it is checked if a specific circuit is available, which then takes precedence over the generic implementation. This also allows resource trade-offs, as the plug-ins can be designed to make use of slices, DSPs, or table look-ups in block memory. The generic $\log_2(q) \times \log_2(q)$-bit multiplier is instantiated using the Xilinx provided multiplication IP cores where an appropriate core is selected considering the size of q (maximum 64×64-bit). The usage of these cores also allows trade-offs between DSPs and logic slices as well as between performance and latency.

4.3.2 Modular Reduction

For modular reduction several options and algorithms have been developed where Barret reduction [Bar86], Montgomery reduction [Mon85], and reductions based on special primes [VOMV96] are the most well known (see [DS07] for a comparison). For the realization of the NTT we have evaluated some of these methods on certain primes and also introduce a generic implementation. Note that some choices for the prime q would lead to simpler reduction circuits than others but that the requirement that $q \equiv 1 \mod 2n$, where n is a power of two, restricts the number of available primes, especially for small values of q.

Generic primes

The purpose of our generic solution is to enable usage of the NTT for arbitrary values of q with support for a high clock frequency. In this case we do not aim for high efficiency but want to support quick prototyping and testability. For this purpose we instantiate a pipelined chain of

---

[3] Note that the efficient NTT algorithms covered in Section 2.4.2 only require access to ω^0 to $\omega^{\frac{n}{2}-1}$ and ω^{-0} to $\omega^{-(\frac{n}{2}-1)}$, respectively.

comparison and conditional subtraction circuits using the VHDL `for generate` statement for a q fixed at synthesis time. This resembles Algorithm 20 where the loop body is unrolled, each left shifts of q precomputed, and x registered after every loop iteration.

Algorithm 20 Generic Reduction of $x \bmod q$

```
Input: $x \in \{0, (q-1)^2\}$
  $l \leftarrow \lceil \log_2((q-1)^2) \rceil - \lceil \log_2(q-1) \rceil$
  for $i = l-1$ downto 0 do
    if $x \geq (q \ll i)$ then
      $x \leftarrow x - (q \ll i)$
    else
      $x \leftarrow x$
    end if
  end for
  return x
```

Fermat primes

Fermat primes are prime numbers of the form $2^{2^z} + 1$ for an integer z and thus satisfy the requirement that $q \equiv 1 \mod 2n$ in most cases. However, the only known Fermat primes are $3, 5, 17, 257, 65537$ where only 65537 seems a suitable choice for lattice-based encryption and signature schemes[4], e.g., assuming $n = 256$, $n = 512$, or $n = 2048$. The modular reduction for Fermat primes is easy to compute. For $q = 65537$ it holds that $2^{16} \mod 65537 \equiv -1$ and thus for an integer x in $[0, 2^{32})$ it holds that $x' = x \mod q \equiv x_{15,\ldots,0} + (q - x_{31,\ldots,16})$ where only one final subtraction of q is necessary in case $x' \geq q$.

Barret reduction

Barret reduction [Bar86] is a common algorithm for a fixed modulus. It can be used to compute $r = x \bmod q$ and relies on the fact that $r = x - q \left\lfloor \frac{x}{q} \right\rfloor$. However, the computation of $\frac{x}{q}$ would normally require a floating or fixed point division which by itself is very expensive. But it is possible to precompute $u = \left\lfloor 2^{2\lceil \log_2 q \rceil} (\frac{1}{q}) \right\rfloor$ so that the reduction is $r = x - q((ux) \gg 2\lceil \log_2 q \rceil)$. As q and u are known in advance the multiplication can be realized as shifts-and-adds and does not require an expensive multiplier or a DSP. To evaluate Barret reduction we have implemented a reduction unit and PE for $q = 7681, q = 12289$, and $q = 8383489$. The algorithm for $q = 12289$ is given in Algorithm 21 and the algorithm for $q = 8383489$ is given in Algorithm 22. For the $q = 8383489$ case the multiplication by q was optimized in the sense that we write q in non-adjacent form (NAF) [CFA06, Definition 9.13] as $q = 2^{23} - 2^{13} + 2^{12} - 2^{10} + 1$ in order to achieve a low hamming weight representation and thus a minimal number of additions. The final implementation of our Barret reducers was done using Xilinx Vivado High-Level Synthesis (HLS) [Xil14]. The tool allows the generation of VHDL code out of the description of the circuit using the C or C++ programming language and the description for $q = 7681$ is given in Figure 4.2.

---

[4] For the lattice-based hash function SWIFFT [LMPR08] $q = 257$ and $n = 64$ has been used.

Algorithm 21 Reduction x mod 12289

```
Input: $x \in \{0, (12289-1)^2\}$
  $t \leftarrow ((x \ll 1)+(x \ll 4)+(x \ll 6)+(x \ll 8)+(x \ll 10)+(x \ll 12)+(x \ll 14)) \gg 28$;
  $t \leftarrow x-(t \ll 12+t \ll 13+t)$
  if $t \geq 12289$ then
    $t \leftarrow t-12289$
  end if
  return t
```

Algorithm 22 Reduction x mod 8383489

```
Input: $x \in \{0, (8383489-1)^2\}$
  $t \leftarrow (x \ll 23+(x \ll 12)+x \ll 10+x) \gg 46$;
  $t \leftarrow x-(t \ll 23-(t \ll 13)+t \ll 13-(t \ll 10)+t)$
  if $t \geq 8383489$ then
    $t \leftarrow t-8383489$
  end if
  return t
```

```
ap_uint<13> vivado_hls_mod_7681(ap_uint<26> val){
        ap_uint<45>  temp = val;

        temp = ((temp<<5) + (temp<<9)+(temp<<13)) >> 26;
        val = val - ((temp<<13) - (temp<<9) + temp);

        if (val >= 7681)
                val = val - 7681;

        return val;
}
```

Figure 4.2: Barret reduction modulo 7681 implemented in Vivado HLS.

Prime 8383489

For the reduction modulo $q = 8383489$ we rely on an idea by Solinas [Sol99]. Obviously, the value $2^{23} \mod 8383489 = 5119$ is small. By applying Solina's idea to reduce a binary number $u = x_{45...0} \mod q$ we write $c = 2^{23}x_{45...23} + x_{22...0} \mod q = 5119 \cdot x_{45...23} + x_{22...0}$. Using this we reduced the result of the multiplication by 10 bits (see Figure 4.3 for the complete block diagram). Applying this trick iteratively for three times and after some subtractions we finally get the reduced value u. In addition to that, the multiplication by the constant $5119 = 2^{12} + 2^{10} - 1$ can be implemented very efficiently with two simple shifts, one addition, and one subtraction.

4.3.3 The NTT and Memory Access Restrictions

As previously explained, our polynomial multiplication unit uses two distinct RAMs, each of size $n\lceil \log q \rceil$ bits, to store the coefficients of the input polynomials $\mathbf{a} \in \mathcal{R}_q$ and $\mathbf{b} \in \mathcal{R}_q$. All arithmetic operations are carried out by the shared PE. When coefficients $\mathbf{a}[i]$ or $\mathbf{b}[i]$ with $0 \leq i < n$ are loaded sequentially into the core they are multiplied by ψ^i and stored at bit-reversed locations in one of the RAMs. The twiddle factors ω^k for $0 \leq k < \frac{n}{2}$ as well as ψ^j for $0 \leq j < n$ (and their inverse counterparts) are stored in a read-only $2n\lceil \log q \rceil$-bit block memory. The NTT is realized iteratively, in-place, and uses a decimation-in-time (DIT) algorithm to compute the

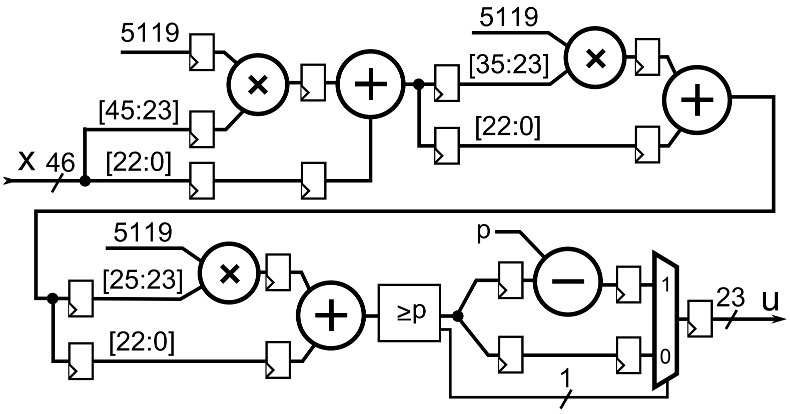

Figure 4.3: Pipelined reduction modulo $q = 8383489$ where multiplication by the constant 5119 is realized with shift-and-adds.

NTT of a polynomial in one of the RAMs. After the coefficient-wise multiplication the result is stored in the second RAM while the first RAM still contains the transformed coefficient. After an inverse NTT, which is a forward NTT with different constants, each j-th coefficient of the second RAM is sequentially multiplied by the scalar n^{-1} and by ψ^{-j} for $0 \leq j < n$. Note that the butterfly structure depicted in Figure 4.1 needs two inputs and produces two outputs during one clock cycle. However, block-RAM on FPGAs usually supports dual-port access so that only two values can be written and/or read independently in one clock cycle. Therefore, we rely on an observation by Pease [Pea68] that the parities of the address lines requesting the inputs and outputs for the butterfly always differ to realize conflict free addressing.

The interface of our multiplication-only core consists of a port for coefficients of **a** and **b** and the result **c** is then outputted on another port. Configuration of a top-level instantiation just requires the specification of $n, q, \psi, \psi^{-1}, n^{-1}$ in a VHDL `generic map` statement. All needed tables are computed by the synthesizer and mapped into a block RAM using inference[5] [Xil09b, Chapter 3] and thus no external script is required.

4.4 A Microcode Engine for Ideal Lattice-Based Cryptography

In practical lattice-based cryptography, polynomial multiplication is the most expensive, but not the only operation. Usually, sampling of noise polynomials (uniform or Gaussian), accessing and storage of temporary or preloaded polynomials (e.g., secret/public keys), and addition or subtraction of polynomials is also required. Moreover, in practice a straightforward polynomial multiplication $\mathbf{c} = \mathsf{INTT}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$ is usually not required or would not fully utilize the power of the NTT. The reason is that we want to support cases in which one coefficient is fixed and already stored in NTT format (e.g., a secret or public key) and another one is randomly sampled. Moreover, in case of a computation like $\mathbf{c}_1 = \mathbf{a}_1\mathbf{e}$ and $\mathbf{c}_2 = \mathbf{a}_2\mathbf{e}$ it is only necessary

---

[5] Inference is the instantiation of a block RAM (or other hard macro) from an abstract VHDL description instead of an instantiation using an (interactive) vendor tool.

to transform $\mathbf{e}$ once into the NTT format (and if possible $\mathbf{a}_1, \mathbf{a}_2$ would also be stored in NTT representation).

Table 4.1: Basic instruction set of the proposed ideal lattice microcode engine.

Command	Op 1	Op 2	Cycles	Explanation
C-REV-{A/B}	$\{2, 3+k\}$	-	$n+\epsilon$	Loads a polynomial into register R$\{0/1\}$ of the NTT engine, performs the bit-reversal step and multiplies with NTT constants.
C-NTT-{A/B}	-	-	$\frac{n}{2}\log n+\epsilon$	Executes the NTT on register R$\{0/1\}$.
C-PW-MUL	-	-	$\frac{3}{2}n+\epsilon$	Point/coefficient-wise multiplication of registers R0 and R1. The bit reversed result is stored in register R1.
C-INTT	-	-	$\frac{n}{2}\log n+\epsilon$	Executes the inverse NTT on register R1.
C-INV-PSI	-	-	$n+\epsilon$	Multiplies coefficients in R1 and multiplies with NTT constants.
C-INV-N	-	-	$n+\epsilon$	Multiplies coefficients in R1 with n^{-1}.
C-ADD	$\{0, 3+k\}$	$\{0, 3+k\}$	$n+\epsilon$	Adds two polynomials ($R(op1) \leftarrow R(op1) + R(op2)$)
C-SUB	$\{0, 3+k\}$	$\{0, 3+k\}$	$n+\epsilon$	Subtracts two polynomials ($R(op1) \leftarrow R(op1) - R(op2)$)
C-MOV	$\{0, 3+k\}$	$\{0, 3+k\}$	$n+\epsilon$	Moves a polynomial from one to another register ($R(op1) \leftarrow R(op2)$).
C-WAIT-SAMPLER	-	-	ϵ	Waits until the sampler has buffered more than n coefficients.
C-NTT-GP-MODE	-	-	ϵ	Export special purpose NTT registers as general purpose registers until the next NTT operation.

The runtime depends mainly on the dimension n. The user can configure the instantiation of k general purpose registers which start at address 4 as registers R0 to R3 are special purpose registers. Note that between every instruction a certain number of wait cycles ϵ is required in order to clear the pipeline and reconfigure the switch matrix (the depth of the pipeline depends, e.g., on q).

Thus we have used the NTT multiplier implementation described in the previous section as core component of a synthesis-time configurable microcode engine that offers the user fine grained access to NTT operations and general purpose instructions like random sampling, addition, and subtraction. The microcode engine has to be configured by the instantiating component using the VHDL `generic map` statement and the required constants are $n, q, \psi, \psi^{-1}, n^{-1}$ (assuming the NTT supporting fast negative-wrapped convolutions exists for these parameters). Moreover, the user can choose to instantiate a configurable number of registers k with a variable coefficient width ($width_i$) that can be initialized with precomputed data during synthesis (path to a file given in $init_i$) for $i \in \{0, k-1\}$. The variable coefficient width allows potential saving of block memories in case of temporary values (e.g., noise) or secret and public keys having coefficients smaller than q.

The instruction set of our microcode engine is given in Table 4.1. The most important instructions supported by the processor are the iterative forward (C-NTT) as well as the inverse transform (C-INTT) which take $\approx \frac{n}{2}\log_2 n$ cycles. Other instructions are for example used for the bit-reversal step (C-REV), point-wise multiplication (C-PW-MUL), addition (C-ADD), or subtraction (C-SUB) – each consuming $\approx n$ cycles. Note that the sampler and the I/O port are just treated as general purpose registers. Thus no specific I/O or sampling instructions are necessary and for example the C-MOV command can be used. Note also that the implementation of the NTT is performed in-place and commands for the inverse transformation (e.g., C-PW-MUL or C-INTT) modify only register R1. Therefore, after an inverse transform a value in R0 is still available. The C-WAIT-SAMPLER instruction has to be executed before accessing the sampler port to ensure that enough data is available in the internal buffer of the sampler (usually n). This is mandatory since some instructions (e.g., C-ADD or C-SUB) expect to read values continuously for n cycles after invocation.

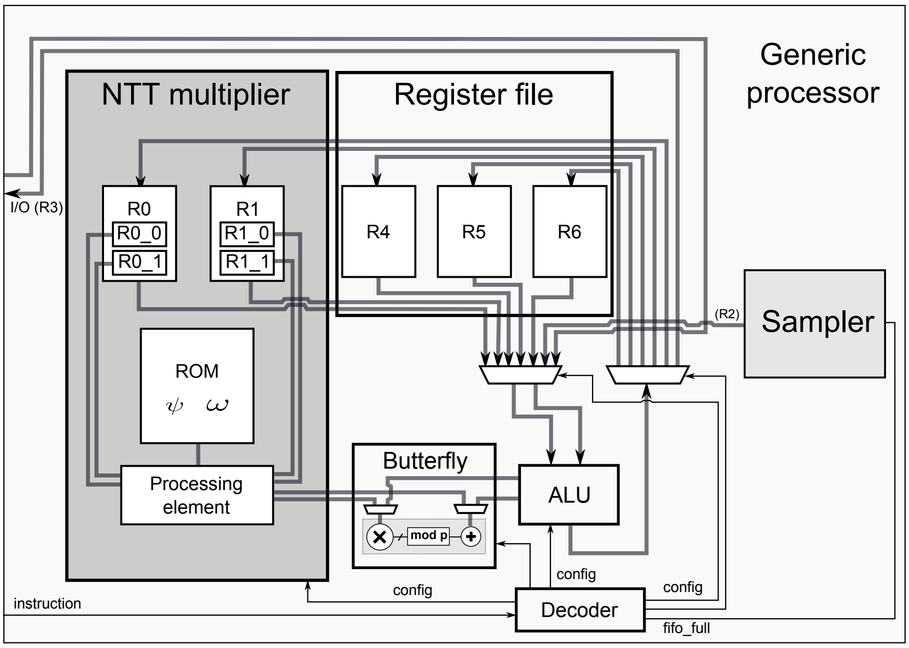

Figure 4.4: Architecture of our implementation of the microcode engine showing a particular instance of our generic lattice processor with three additional registers R4-6.

The datapath of our engine is depicted in Figure 4.4. Four registers are fixed where register R0 and R1 are part of the `NTT multiplier` block, while the sampler is connected to register R2. Register R3 is exported to upper layers and operates as I/O port. Each additional configured register (with an index starting from R4) can hold a polynomial with n elements of size $\log_2 q$ or less (depending on $width_i$). A multiplexer connects registers to the `ALU` and the external interface and is designed to process statements in two-operand form like $R1 \leftarrow R1 + R2$. This is also the main reason for the separate instantiation of registers in block RAMs in contrast to storage in a large continuous memory space as one register has to be read and written at the same time and another one read. All these additional registers are placed inside of the `Register file` component. A decoder unit is responsible for interpreting instructions that configure multiplexers and for determining whether the `ALU` has to be used (C-SUB, C-ADD, C-MOV) or if NTT specific commands need to invoke the `NTT multiplier`. To improve resource utilization of the overall system, the butterfly unit of the NTT core is shared between the NTT multiplier and the ALU.

4.5 Implementation of Schoolbook Multiplication

To compare the proposed NTT multiplier with the naive approach we have developed a basic schoolbook polynomial multiplier according to the description in Section 2.4.1. For easier comparison of the resource consumption we reuse the arithmetic already implemented in the processing element (see Section 4.3.1 and Figure 4.1). Our constant time schoolbook implementation uses $\approx n^2 + 2n$ cycles with the constant $2n$ being attributed to loading of the two inputs while outputting of the result is performed directly in the final round. The multiplier needs three distinct $n \times \lceil \log_2(q) \rceil$-bit memories to store the two input polynomials $\mathbf{a}$ and $\mathbf{b}$ and the temporary result. The implementation results for selected parameters are given in Table 4.5.

An advantage of the schoolbook approach and the implementation is that it is more versatile than the NTT core as it does not place restrictions on the modulus q and the degree of polynomials n. However, in this section we specifically focus on dense polynomials of equal size and do not exploit different densities of input polynomials (by skipping the addition of lines if a zero is detected). However, we describe an implementation of a schoolbook multiplier that exploits special characteristics of input polynomials for the implementation of RLWEenc in Section 5.4. A schoolbook multiplier that exploits the sparseness of input polynomials in the context of lattice-based signature schemes is provided in Section 7.3.2.

4.6 Results and Comparison

In this section we present synthesis results for the processing element (PE), the proposed NTT multiplier, the schoolbook multiplier as reference, and the NTT microcode engine instantiated with a uniform sampler. All results were obtained post place-and-route (PAR), generated with Xilinx ISE 14.7, and implemented stand-alone on the medium-cost Spartan-6 LX45 (xa6slx45fgg484-3) where the ATHENa tool was used for design exploration [GKA+10].

4.6.1 Processing Element

The results for different implementations of the processing element are given in Table 4.3. As the multiplication is performed with the help of DSP cores, the usage of LUTs and FFs is dominated by the modular reduction. It can be seen that the Barret reduction implemented in Vivado HLS outperforms the generic approach. On the other hand, the special reduction for $q = 65537$ requires less area than the Barret reduction for the smaller prime $q = 12289$. Due to the usage of a large number of FF the generic reduction allows higher clock frequencies than the other approaches.

Table 4.3: Resource consumption and performance results for different instantiations of the PE.

q	Algorithm	Slice	LUT	FF	DSP	BRAM9	MHz
7681	Generic	286	699	420	1	0	293
7681	Barret	68	192	103	1	0	175
12289	Generic	245	782	473	1	0	298
12289	Barret	111	362	235	1	0	208
65537	Fermat	93	275	292	1	0	310
65537	Fermat	83	208	231	1	0	404
8383489	Special	190	603	602	4	0	250
$2^{57}+25 \cdot 2^{13}+1$	Special	401	1195	1095	4	0	171
1061093377	Special	520	1492	1785	16	0	138

4.6.2 Polynomial Multipliers

In Table 4.4 we provide the performance and resource consumption of the polynomial multiplier instantiated on a Spartan-6 FPGA (without the overhead caused by the microcode engine). The general performance of the polynomial multiplication mainly depends on the runtime of the NTT core which requires $\approx \frac{n \log(n)}{2}$ cycles for a forward or an inverse transformation as the butterfly step executes the inner loop of Algorithm 1 every cycle. Therefore, by executing two forward NTTs, multiplying the polynomials component-wise and then executing an inverse NTT we multiply two polynomials in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ in $\approx 3 \cdot (\frac{n \log(n)}{2}) + 5.5n$ cycles. The additional $5.5n$ cycles are introduced by loading of coefficients ($2n$), component-wise multiplication ($1.5n$) and multiplication of the final result by powers of ψ^{-1} and by n^{-1} ($2n$). The resource consumption is mainly influenced by the area required for the instantiation of the PE (see Table 4.3). Additionally, for larger values of n the address path becomes larger. For our comparison we just consider a scenario in which two dense polynomials (very few zero coefficients) with coefficients in the range from zero to $q-1$ are multiplied. By one BRAM we denote a full 18K-bit block RAM which can be split into two independent 9K-bit block RAMs (denoted as 0.5 BRAM).

Our results show that the NTT can be used to achieve the often claimed quasi logarithmic runtime for ideal lattice-based cryptography. Moreover, our design scales well as it support NTTs for polynomials with large dimensions (e.g., $n = 4096$). The clock frequency mainly depends on the critical path of the PE and is rather independent of the dimension n. More performance results for the microcode engine for parameters supporting RLWEenc as well as GLP and BLISS can be found in Chapter 5 and Chapter 7, respectively.

To benchmark the multiplier for parameter sets that require large moduli, we have instantiated the design with parameters of a somewhat homomorphic encryption scheme (SHE) [NLV11]. It is based on ideal lattices with the RLWE problem as hardness assumption [LPR10b] and allows the computation of a fixed amount of additions and multiplications on encrypted data (depending on the parameters). Encryption requires two multiplications in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$ which make up for most of the runtime. The number of coefficients of the used polynomials ranges from $n = 512$ to $n = 131072$. A reasonable set is ($n = 1024$,$q = 1061093377$) for the homomorphic computation of the mean and ($n = 2048$, $q = 2^{57} + 25 \cdot 2^{13} + 1 = 144115188076060673$) for the computation of the variance on encrypted data. In Table 4.4 the performance and resource consumption for

Table 4.4: Resource consumption and performance results for our NTT-based polynomial multiplier.

App.	n	q	LUT	FF	Slice	DSP	BRAM9	MHz	Cycles	Mul/s
Fermat	128	65537	889	1116	375	1	5	211.149	2,342	90,158
Fermat	256	65537	872	1158	372	1	6	203.087	4,774	42,540
Fermat	512	65537	923	1196	425	1	8	197.707	10,014	19,743
Fermat	1024	65537	974	1231	408	1	13	214.823	21,278	10,096
Fermat	2048	65537	1064	1339	439	1	25	217.061	45,326	4,789
Fermat	4096	65537	1094	1378	470	1	50	218.15	96,526	2,260
LWE [GFS+12]	256	7681	831	978	376	1	6	212.269	4,790	44,315
LWE [GFS+12]	512	12289	1080	1066	418	1	7	183.184	10,161	18,028
SHE [NLV11]	1024	1061093377	2298	2486	912	4	23	172.058	21,405	8,038
SHE [NLV11]	2048	$2^{57}+25\cdot2^{13}+1$	3104	4289	1321	16	86	135.704	45,453	2,986

Table 4.5: Resource consumption and performance results for our schoolbook algorithm-based polynomial multiplier.

App.	n	q	LUT	FF	Slice	DSP	BRAM9	MHz	Cycles	Mul/s
Fermat	128	65537	417	514	161	1	3	301	16,670	18,085
Fermat	256	65537	425	527	166	1	3	308	66,078	4,662
Fermat	512	65537	449	540	166	1	3	293	263,198	1,113
Fermat	1024	65537	464	603	192	1	6	288	1,050,654	274
Fermat	2048	65537	483	616	197	1	12	286	4,198,430	68
Fermat	4096	65537	504	630	206	1	27	303	16,785,438	18
LWE [GFS+12]	256	7681	447	434	154	1	3	244	66,078	3,689
LWE [GFS+12]	512	12289	590	475	199	1	3	182	263,210	690
SHE [NLV11]	1024	1061093377	1503	1608	545	4	12	181	1,050,854	172

these parameter sets is detailed. With a clock frequency of 161 MHz we can compute 3,542 polynomial multiplications in one second (0.28 ms per multiplication) for the larger parameter set. In [NLV11] it is reported that the same operation takes 11 ms on a 2.1 GHz dual core processor when implemented in Magma and thus we achieve a speed-up by a factor of 39 for this operation that contributes heavily to the overall runtime.

Results for our schoolbook polynomial multiplier are given in Table 4.5. The design requires very roughly only half of the slices of the NTT multiplier. However, the used amount of BRAMs is larger and the performance is worse than the NTT multiplier for any value of n. As a consequence, it appears that the NTT is the overall better choice for the implementation of high-performance multiplication for dense and large polynomials in ideal lattice-based cryptography.

4.6.3 Comparison with Related Work

In this section we compare our implementation of polynomial multiplication with relevant implementations on reconfigurable hardware and highlight the different approaches. Moreover, we revisit follow-up works by other researchers.

Comparison with Previous Implementations

In [GCHB12, GCB13] Györfi et al. proposed an implementation of the SWIFFT(X) hash function [LMPR08] which uses the FFT/NTT as a core primitive. For their implementation they utilize the fact that the small modulus $q = 257$ is fixed. This allows the usage of look-up tables instead of DSP-based arithmetic. Furthermore, they utilize the diminished-one number system in order to represent numbers modulo $q = 2^k + 1$. The resource consumption is 3,639 slices and 68 BRAMs on a Virtex-5 LX110T FPGA and their pipelined implementation is able to compute the FFT in one clock cycle per sample with a latency of $\log_2(n)$ at a frequency of 150 MHz. Note that a fair comparison with our results is not possible as the parameters of SWIFFT $(n = 64, q = 257)$ are much smaller parameter than those we have considered. A similar parameter set $(n = 128, q = 257)$ was used for an implementation of the SPRING [BBL$^+$14] pseudo random generator (PRG) by Brenner et al. [BGL$^+$14] that also utilizes the FFT/NTT.

A complete and fully functional reconfigurable hardware implementation of ring-LWE encryption is given in [GFS$^+$12] with a special emphasis on efficient Gaussian sampling and the parallel implementation of the NTT. The result of a polynomial multiplication is available in $\mathcal{O}(\log n)$ time instead of $\mathcal{O}(n \log n)$ required by our iterative implementation. However, due to this parallel approach the resource consumption of the implementation is very high, especially as no internal block RAMs or DSPs have been used. For a comparison of our implementation of ring-LWE encryption based on the multiplier presented in this chapter with [GFS$^+$12] we refer to Section 5.5.

Review of Follow-up Work on Polynomial Multiplication

In this section we discuss follow-up works by other authors [APS13, CMV$^+$14, RVM$^+$14] works who directly refer to the original publication of the proposed polynomial multiplier [PG12] or the microcode engine [PG13] for comparison.

Polynomial Multiplier by Aysu et al. [APS13]. A polynomial multiplier supporting a variable n but fixed $q = 2^{16} + 1 = 65537$ for simple modular reduction (see Section 4.3.1) was proposed by Aysu et al. in [APS13]. Their implementation is a pure polynomial multiplier designed to compute $\mathbf{c} = \mathbf{ab} \in \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ and does not target a specific scheme or parameter set. The implementation takes two polynomials $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ as input and transforms them in parallel into the NTT domain, performs point-wise multiplication and the inverse transform to obtain the result $\mathbf{c}$. The main difference to our work is a simplified state logic which stems from a rewriting of Algorithm 1 and the ability to compute powers of ω and ω^{-1} (twiddle factors) as well as powers of ψ and ψ^{-1} on-thy-fly. The biggest challenge when using this approach instead of a table is the required short pipeline as subsequent computations depend on the previous results. This approach thus trades the ROM used in our implementation against one or two additional multipliers (realized by DSPs) and reduction circuits (which are rather cheap for

$q = 65537$). In the 2DSP architecture one DSP is used to realize the processing element (as in Figure 4.1) and one DSP is used for twiddle factor generation. In the 3DSP architecture an additional DSP is used to compute twiddle factors which allows a longer pipeline and thus a higher clock frequency. The block memory requirements of the implementation by Aysu et al. are lowered, as no ROM is necessary to store powers of $\omega, \omega^{-1}, \psi, \psi^{-1}$. Moreover, two input coefficient pairs are stored adjacent in one block RAM address. As each coefficient is 17 bits, two adjacent coefficients require 34 bits (maximum is 36 bits on a Spartan-6). As a consequence, the numbers of BRAM reads is decreased and thus the RAM does not have to be split in even and odd parity RAMs as in our implementation which leads to an efficient forward transformation. Additionally, as BRAMs are a discrete element that can usually not be shared easily, one large and fully utilized BRAM ($n \times 2\log_2 q$ bits) is usually more efficient than two small and underutilized BRAMs (each $n \times \log_2 q$ bits) that have to be divided further into an odd and an even parity BRAMs (each $\frac{n}{2} \times \log_2 q$ bits), especial when the BRAMs are not completely filled and thus waste unused memory[6]. However, this approach leads to so called "bubbles" during the inverse transformation as not enough memory ports are available to feed the two inputs and to write back the two outputs of the butterfly and thus every second cycle the PE is unused. As a consequence, the implementation proposed in [APS13] requires $2n(\log_2(n)) + 7n$ cycles[7] which is more than the $\approx \frac{3}{2}(n \log n) + 5.5n$ of our implementation for common choices of n. As an example, for $n = 512$ the implementation of Aysu et al. needs 11,264 cycles while our work requires $\approx$ 9,728 cycles.

Polynomial Multiplier by Chen et al. [CMV+14]. An NTT polynomial multiplier to compute $\mathbf{c} = \mathbf{ab} \in \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ targeting RLWEenc [RS10] and somewhat homomorphic encryption (SHE) [NLV11] parameter sets of [PG12] has been given by Chen et al. in [CMV+14]. Their design goal is high speed and low latency which is achieved by using two PEs and two additional integer modular multipliers. During the forward transformation one PE is used for each input polynomial $\mathbf{a}, \mathbf{b}$ and during the inverse transformation both PEs are used to compute $\mathbf{c}$ in parallel. The two additional multipliers are responsible for the multiplication of polynomials by powers of ψ and ψ^{-1} as well as point-wise multiplication. To achieve a high frequency the constant geometry FFT [Pea68] is used which simplifies address generation as the same datapath can be used in every FFT stage. Additional optimizations include the precomputation of $n^{-1}\psi^{-i}$ (instead of just ψ^{-i}) for $0 \leq i < n$ and the observation that in the first FFT stage multiplications by $\omega^0 = 1$ can be removed. The authors further introduce a method to select primes that lead to an efficient implementation of modular reduction based on the techniques introduced by Solinas [Sol99] that were also used previously for efficient modular reduction in [PG12, GLP12].

Microcode Engine by Roy et al. [RVM+14]. A resource efficient NTT multiplier controlled by a microcode engine for the implementation of the RLWEenc parameter sets ($n = 256, q = 7681$) and ($n = 512, q = 12289$) has been proposed by Roy et al. in [RVM+14]. It basically supports

---

[6] As an example, for $n = 256$ the implementation of Aysu et al. requires a 256×34 BRAM (one BRAM9) while our implementation needs four 128×17 BRAMs (four BRAM9) which are only filled up to 24 %. However, this effect is less important for larger values of n as the utilization increases.

[7] We assume from the description in the paper that the parallel forward transform of two polynomials requires $n \log_2(n)$ cycles and the inverse transform also $n \log_2(n)$ cycles where 50 % cycles required for the inverse transform are wait cycles.

the same operations as the microcode engine discussed in Section 4.4. The design consists of an NTT multiplier with access to a register file and offers instructions for storing, loading, adding, and sampling of polynomials from a Gaussian distribution. The proposed multiplier design builds on top of the work by Aysu et al. [APS13]. The most important improvement is a technique to remove the bottleneck when accessing the BRAM holding polynomial coefficients where two coefficients are stored pairwise in one memory address. Additionally, on-the-fly computation of twiddle factors is optimized by reordering of the NTT algorithm and storage of few intermediate powers of ω and ω^{-1}. From the description we assume the costs of a polynomial multiplication to be in the order of $\frac{3}{2}\log_2 n + 2n$ cycles as the only operations with linear complexity are point-wise multiplication and the final multiplication of coefficients by $n^{-1}\psi^{-i}$ for $0 \leq i < n$.

Comparison of Proposed Polynomial Multipliers

A comparison of the previously discussed polynomial multipliers for selected parameter sets and design variants is given in Table 4.6. Note that due to its high resource consumption, lack of a detailed analysis, and embedding into an RLWEenc core we do not consider the polynomial multiplier by Göttert et al. [GFS+12] in this section. The given results show that the authors of the `APS` design were able to reduce the number of required slices by approximately 400 and also use less block RAMs. The `CMVRCPV` design focuses on speed and low latency by introducing more parallelism into the NTT computation, achieves a high clock frequency, but is currently the largest one. The listed resource consumption of the `RVMCV` accounts for a full implementation of RLWEenc and is still only as large as our stand-alone polynomial multiplier.

Table 4.6: Comparison of the polynomial multiplier designs of Aysu et al. [APS13] (`APS`), Chen et al. [CMV+14] (`CMVRCPV`), and Roy et al. [RVM+14] (`RVMCV`).

Design	App.	n	q	LUT	FF	Slice	DSP	BRAM9	MHz	Cycles	Mul/s
`APS`	-	128	65537	592	547	241	2	1	244	2688	90671
`APS`	-	256	65537	608	527	247	2	1	231	5888	39205
`APS`	-	512	65537	632	535	256	2	2	224	12800	17517
`APS`	-	1024	65537	653	543	260	2	4	222	27648	8029
`APS`	-	2048	65537	670	552	270	2	8	217	59392	3648
`APS`		4096	65537	676	562	265	2	16	211	126976	1660
`CMVRCPV`	SHE	1024	536903681	6689	-	2112	4	8	211	7967	26458
`CMVRCPV`	SHE	2048	$2^{57}+25\cdot2^{13}+1$	14105	-	4406	12	50	208	17402	11959
`RVMCV`	RLWE	256	7681	1349	860	-	1	2	313	3584	87333
`RVMCV`	RLWE	512	12289	1536	953	-	1	3	278	7936	35030

Note that the design `RVMCV` is a full implementation of RLWEenc and thus also contains a Gaussian sampler and additional memory.

4.7 Conclusion and Future Work

In conclusion, it can be seen that the polynomial multiplier proposed in this chapter was a huge improvement over previous work [GFS+12] and applicable for the implementation of lattice-based encryption [PG13] and signature schemes [PDG14a, GLP15]. Moreover, our initial design was a first approach that has been extended by follow-up work. Examples are designs that improved speed [CMV+14] or area consumption [APS13, RVM+14] or that were optimized for certain application scenarios. However, we would like to note that in practice a multiplier has to be flexible enough to deal with already transformed inputs (e.g., public keys, polynomials used twice) in order to exploit the full potential of the NTT. Thus a pure comparison of the speed for a multiplication $\mathbf{c} = \mathbf{ab}$ can be misleading as it heavily depends on the application scenario and a fast forward transformation (e.g., using two PEs) is probably more useful than two parallel but slower forward transformations (where one PE is used per transformation). Additionally, a multiplier should be embedded into a configurable microcode engine like [RVM+14] as most practical applications do not only require polynomial multiplication but also addition and subtraction, which can be handled by the arithmetic in the PE.

When reviewing some design decisions, given the availability of follow-up work and applications, the general multiplier design still seems to be sound, which is not surprising as it has been already used for numerous FFT implementations. However, the biggest structural deficit of the microcode engine from Section 4.4 (addressed in [RVM+14]) is the distinction between the two internal NTT-enabled registers and external registers for temporary values which adds additional complexity and requires time for memory transfers between these two register spaces. Moreover, our implementation allows operations in two operand from (e.g., R2=R2+R5). This is possible as each register is realized in a separate block RAM (two ports are required). However, a different approach with a continuous memory space but more restrictions on the operations would offer much better utilization and simpler access logic. From an implementation point of view, the use of inference of block memories turned out to be convenient and allowed simple configuration using the VHDL `generic` statement in the top-level component. However, inferred memories are usually larger compared to memory generated with the less flexible core generator. Additionally, our implementation contains too many pipeline stages for small parameter sets and too few for the larger ones.

Certainly, there is still room for improvement in several directions; usage of more PEs and improved memory organization could lead to even faster implementations, resource consumption could be further lowered, or applications like homomorphic cryptography could profit from designs supporting very large values of n and q. Another possible future work would be to realize the optimizations of the NTT discussed in Section 6.2 on reconfigurable hardware. However, from an application perspective it seems that lattice-based cryptography is already extremely fast (see [RVM+14] and Chapter 5) and the question arises whether application that used public-key encryption or signatures in hardware could really profit from even higher speed enabled by faster polynomial multipliers. Additionally, we would like to note that this research was inspired to a large extend by literature on the implementation of the FFT for signal processing and that most techniques used in [APS13, RVM+14, CMV+14] were also previously proposed (see e.g., publications like [Pea68, Ber69] that date back into the years 1968 and 1969, respectively). This is certainly an advantage for lattice-based cryptography as a lot of implementation techniques are already available. They just have to be evaluated and adapted for the specific use case but not invented anymore. We additionally see huge opportunities for collaboration between the lattice-based cryptography community and the signal processing community on this issue.

Chapter 5

Implementation of Ring-LWE Encryption on Reconfigurable Hardware

In this chapter we provide two implementations of the RLWEenc *public-key encryption scheme on reconfigurable hardware. By leveraging our microcode engine (see Chapter 4) and its fast NTT-based polynomial multiplication routines we realize a high-speed implementation that significantly reduces the time-area product compared to previous work. We also present a design that aims at minimal area consumption and still achieves performance that seems to be sufficient for most applications. Our work shows that public-key encryption based on the* RLWE *problem can be realized efficiently and fast on reconfigurable hardware. Additionally, we cover some techniques to reduce decryption errors and ciphertext expansion. The high-speed implementation presented in this chapter is based on [PG13] and the low-area implementation appeared in [PG14]. Some background material from [POG15a] is also used.*

Contents of this Chapter

5.1 Introduction

Public-key encryption (PKE) is a fundamental asymmetric cryptographic primitive and plays an important role in a huge number of applications or security protocols. The first popular lattice-based PKE scheme is NTRUEncrypt which was proposed by Hoffstein, Pipher, and Silverman [HPS98] and operates on polynomials in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n - 1\rangle$. Since its introduction, several works have covered efficient implementation on various platforms [ABF$^+$08, KY09, HVP10] as well as analysis of security properties [HHHW09]. Stehlé and Steinfeld [SS11] also proposed a provably secure NTRUEncrypt variant that is based on ideal lattices and defined in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$. However, the provably secure variant requires rather large parameters and Cabarcas et al. [CWB14] concluded that it is inferior to other lattice-based PKEs.

Alternatively, the LWE problem [Reg05] can also be used in a rather straightforward manner to construct a lattice-based PKE [LP11]. However, it is not clear whether schemes with reductions to hard lattice problems are more efficient than classical schemes or other post-quantum schemes, e.g., NTRU. The currently and arguably most efficient instantiation of a lattice-based PKE with a security reduction is based on the RLWE problem [LPR10b]. We refer to this scheme as RLWEenc [LPR10b, LP11] and introduce it in Section 3.4. From a hardware point of view, RLWEenc appears very interesting as it is conceptually simple and only requires polynomial multiplication, polynomial addition, as well as sampling of polynomials according to a narrow discrete Gaussian distribution. Moreover, it is not covered by any patents (up to our knowledge), allows fast key generation, and already achieves CPA-security in its plain version.

A huge advantage of RLWEenc is that the scheme can be optimized for high performance when using the NTT. The idea is to store secret and public-key polynomials in the NTT domain during key generation, which simplifies the encryption and decryption algorithms. Additionally, one noise polynomial ($\mathbf{e}_1$) is used in two polynomial multiplications ($\mathbf{ae}_1$ and $\mathbf{pe}_1$) and has to be transformed only once into the NTT domain. As a consequence, only a small extension of the microcode engine discussed in Chapter 4 is necessary to realize the scheme on reconfigurable hardware supporting high performance key generation, encryption, and decryption in one core. Moreover, RLWEenc can also be implemented in a more lightweight manner without exploiting the benefits of the NTT but by relying on simple schoolbook multiplication. In this case address generation becomes extremely simple, no precomputed constants like twiddle factors are necessary, and restriction on parameter choice are relaxed as no roots of unity are required. Additionally, a result proposed by Brakerski et al. [BLP⁺13] on parameter selection gives hints that it is also possible to choose a power-of-two modulus, which simplifies modulo reduction.

While performance, simplicity, and security properties of RLWEenc are a convincing argument for further considerations, there are also some drawbacks that need to be addressed by cryptographers and implementers. The biggest issues with RLWEenc and most LWE-based schemes in general are large ciphertexts and that decryption errors can occur for some parameter sets. When selecting parameters, one basically has to make a trade-off between the desired security level, ciphertext size, and an acceptable decryption error rate. As an example, the medium security parameter set RLWEenc-Ia proposed in [GFS⁺12] results in a ciphertext expansion by a factor of 26 for every plaintext bit but has still non-negligible error rates. Moreover, in the plain scheme the plaintext space is limited to n bits where n is the dimension of the used polynomials.

5.1.1 Related Work

The RLWEenc scheme we are focusing on in this chapter has been introduced in [LPR10b, LPR10c, LP11] and is discussed in detail in Section 3.4. Commonly used parameters have been proposed in [LP11, GFS⁺12] and a security analysis of the scheme can be found in [LP11, LN13]. The first FPGA implementation has been provided by Göttert et al. [GFS⁺12]. In [RVM⁺14] Roy et al. presented the currently fastest implementation by using new techniques for faster NTT-based polynomial multiplication. The efficient sampling of polynomials from a narrow discrete Gaussian distribution has been examined in [DG14] and a hardware implementation of a sampler has been proposed in [RVV13]. The RLWEenc scheme is also employed in a recently proposed identity-based encryption scheme (IBE) [DLP14] and is similar to a key exchange protocol [Pei14] that can be used in the transport layer security (TLS) protocol [BCNS15].

5.1.2 Contribution

The first part of the contribution of this chapter is a fast implementation of RLWEenc on reconfigurable hardware that exploits the properties of the NTT. Using the same parameters as in work by Göttert et al. [GFS+12] we improve their results by at least an order of magnitude considering a throughput to area ratio on a similar reconfigurable platform (Virtex-6). On the low-cost Spartan-6 family our core achieves significant performance, namely 39.39 µs to encrypt and 20.13 µs to decrypt a block, with very moderate resource requirements. Moreover, all parts of our implementation have constant runtime and inherently provide resistance against timing attacks.

The second part of the contribution is a low-area implementation that additionally evaluates the consequences of a result by Brakerski et al. [BLP+13] on parameter selection, i.e., the usage of power-of-two moduli. We are able to provide a decryption circuit that can be implemented with only 32 slices, one BRAM, and one DSP block on a Xilinx Spartan-6 FPGA. The encryption module is slightly larger – the reason is that the scheme requires costly sampling from a discrete Gaussian distribution. To implement this operation we combine rejection sampling with Bernoulli trials in order to evaluate the exp() function as proposed in [DDLL13b]. For the necessary standard deviation of $\sigma = 3.33$ we just use 37 slices for the sampler (excluding a random bit source). This approach complements the work of Roy, Vercauteren, and Verbauwhede [RVV13] who also proposed a sampler for the same application scenario.

All in all we provide evidence that public-key encryption based on the RLWE problem can be both fast and area efficient in hardware. Our work demonstrates that lattice-based cryptography is indeed a promising and practical alternative for asymmetric encryption in future real-world systems.

5.2 Optimization of RLWEenc for Efficiency and Correctness

In this section we provide a description of RLWEenc with explicit usage of the NTT. Additionally, we cover and evaluate inexpensive techniques to reduce the error rate and ciphertext expansion.

5.2.1 Application of the NTT

When using the RLWEenc-Ia and RLWEenc-IIa parameter sets as proposed in [GFS+12], the NTT and efficient algorithms for its computation exist as q is chosen to be prime and as it holds that $1 \equiv q \mod 2n$ for n being a power-of-two. Then it is further possible to exploit the general characteristic of the NTT which supports to decompose a multiplication into two forward transforms and one inverse transform. If one coefficient is fixed or needed twice it is then possible to directly store it in NTT representation to save subsequent transformations. In Algorithm 23, 24, and 25 the Gen, Enc, and Dec routines of the RLWEenc scheme are described utilizing this property of the NTT. By SampleGaussPoly we denote a function that samples a polynomial from a discrete Gaussian distribution where all coefficients have standard deviation $\sigma = \frac{s}{\sqrt{2\pi}}$. Additionally, all polynomials $\tilde{\mathbf{r}}_2, \tilde{\mathbf{a}}, \tilde{\mathbf{p}}$ returned by the key generation algorithm are already stored in the NTT domain. The only drawback is that this increases the storage requirement for the secret key from $n\lceil\log_2(\tau\sigma)\rceil$ bits to $n\lceil\log_2(q)\rceil$ bits as $\tilde{\mathbf{r}}_2$ is distributed uniformly and not Gaussian when being stored in the NTT domain. In our proposal, the second part of the

Algorithm 23 RLWEenc Key Generation

```
Precondition: Access to global constant $\tilde{\mathbf{a}} = \mathsf{NTT}(\mathbf{a})$
func $\mathsf{RLWEenc}_{\mathsf{gen}}^{\mathsf{NTT}}()$
    $\tilde{\mathbf{r}}_1 \leftarrow \mathsf{NTT}(\mathsf{SampleGaussPoly}())$
    $\tilde{\mathbf{r}}_2 \leftarrow \mathsf{NTT}(\mathsf{SampleGaussPoly}())$
    $\tilde{\mathbf{p}} = \tilde{\mathbf{r}}_1 - \tilde{\mathbf{a}} \circ \tilde{\mathbf{r}}_2$
    return $(\mathsf{pk}, \mathsf{sk}) = (\tilde{\mathbf{p}}, \tilde{\mathbf{r}}_2)$
end func
```

Algorithm 24 RLWEenc Encryption

```
Precondition: Access to global constant $\tilde{\mathbf{a}} = \mathsf{NTT}(\mathbf{a})$
func $\mathsf{RLWEenc}_{\mathsf{enc}}^{\mathsf{NTT}}(\tilde{\mathbf{a}}, \tilde{\mathbf{p}}, \mu \in \{0,1\}^n)$
    $\tilde{\mathbf{e}}_1 = \mathsf{NTT}(\mathsf{SampleGaussPoly}())$
    $\tilde{\mathbf{e}}_2 = \mathsf{NTT}(\mathsf{SampleGaussPoly}())$
    $\tilde{\mathbf{c}}_1 = \tilde{\mathbf{a}} \circ \tilde{\mathbf{e}}_1 + \tilde{\mathbf{e}}_2$
    $\tilde{\mathbf{h}}_2 = \tilde{\mathbf{p}} \circ \tilde{\mathbf{e}}_1$
    $\mathbf{e}_3 \leftarrow \mathsf{SampleGaussPoly}()$
    $\mathbf{c}_2 = \mathsf{INTT}(\tilde{\mathbf{h}}_2) + \mathbf{e}_3 + \mathsf{Encode}(m)$
    return $c = (\tilde{\mathbf{c}}_1, \mathbf{c}_2)$
end func
```

Algorithm 25 RLWEenc Decryption

```
func $\mathsf{RLWEenc}_{\mathsf{dec}}^{\mathsf{NTT}}(c = [\tilde{\mathbf{c}}_1, \mathbf{c}_2], \tilde{\mathbf{r}}_2)$
    return $\mathsf{Decode}(\mathsf{INTT}(\tilde{\mathbf{c}}_1 \circ \tilde{\mathbf{r}}_2) + \mathbf{c}_2)$.
end func
```

ciphertext $\mathbf{c}_2$ is not transmitted in NTT representation to allow the application of compression techniques introduced in Section 5.2.2. An alternative placement of NTT operations has been shown by Roy et al. [RVM+14]. A comparison of a naive approach, using the NTT just for polynomial multiplication, with our proposal[1] and the proposal by Roy et al. can be found in Table 5.1. Note that Roy et al. describe the key generation without explicit usage of the NTT and just propose to transform the final keys $\mathsf{pk} = (\mathbf{a}, \mathbf{p})$ and $\mathsf{sk} = \mathbf{r}_2$.

5.2.2 Ciphertext Expansion and Decryption Errors

Compared to ECC or RSA the ciphertext expansion of RLWEenc is rather large and despite the threshold decoding (see Section 3.4) infrequent decryption errors appear that lead to flipped bits in the decrypted and decoded message. In this section we show how to reduce the ciphertext expansion and propose methods to deal with decryption errors. One mitigation approach, as discussed in [GFS+12], is to adapt parameters so that the error/noise introduced by the term $\mathbf{e}_1 \cdot \mathbf{r}_1 + \mathbf{e}_2 \cdot \mathbf{r}_2 + \mathbf{e}_3$ gets smaller. However, completely eliminating the error would require large values for q (or smaller σ) and lead to decreased efficiency and increased ciphertext size (or a decreased security level). We thus decided not to change the parameter set but to investigate additional techniques to correct decryption errors with a marginal impact on performance only.

---

[1] Note that in [PG13] a different placement of NTT transformations is used which requires one additional NTT transformation compared to Roy et al. [RVM+14].

Table 5.1: Operation counts for RLWEenc when using the NTT.

Scheme	Operation	NTT	INTT	ADD/SUB	Point-wise
Naive	Gen	2	1	1	1
	Enc	4	2	3	2
	Dec	2	1	1	1
Our work Alg. 23, 24, and 25	Gen	2	0	1	1
	Enc	2	1	3	2
	Dec	0	1	1	1
Roy et al. [RVM+14]	Gen	-	-	-	-
	Enc	3	0	3	2
	Dec	0	1	1	1

An additional benefit of a better mechanism to deal with errors is that it might also allow parameter choices that increase security (roughly speaking a larger s or smaller q).

Reduction of Ciphertext Expansion

Threshold encoding was proposed in [LP11, GFS+12] for RLWEenc to transfer n bits resulting in an inflated ciphertext of size $2n\lceil\log_2 q\rceil$. However, when it is only necessary to transfer a 128-bit AES key, the plaintext space is not fully utilized. As the RLWEenc scheme suffers from random decryption errors this allows to put redundancy in the message to correct those errors. In the following we analyze a simple but effective way to reduce the ciphertext expansion without significantly affecting the error rate. This approach has been previously applied to homomorphic encryption schemes in [BLLN13a, Section 6.4] and [Bra12, Section 4.2] where the idea is basically to remove a certain number of least significant bits of $\mathbf{c}_2$. This works as the lower order bits mostly carry noise but only little information supporting the threshold decoding. We experimentally verified the applicability of this approach in practice with regard to concrete parameters by measuring the error rates for reduced versions of $\mathbf{c}_2$ as shown in Table 5.2 ($u = 1$).

Table 5.2: Bit-error rate for the encryption and decryption of 160,000,000 bytes of plaintext when removing a certain number x of least significant bits of every coefficient of $\mathbf{c}_2$ in RLWEenc.

u	Remove bits x	0	1	2	3	4	5	6	7	8	9	10	11
1	Errors (10^3)	46	46	45.5	45.6	46	46.5	48.6	56.1	94.4	381	5359	135771
	Error rate (10^{-5})	3.59	3.59	3.56	3.57	3.59	3.63	3.80	4.38	7.38	29.81	418.7	10610
2	Errors	26	20	26	27	23	21	21	32	71	957	125796	$44 \cdot 10^6$
	Error rate (10^{-8})	2.03	1.56	2.03	2.11	1.80	1.64	1.64	2.5	5.55	74.7	9830	$34 \cdot 10^5$

The experiment was performed for the parameter set ($n = 256, q = 7681, s = 11.31$) where u is the parameter of the additive threshold encoding (see Algorithm 26) and $\pm\lceil 2s\rceil$ the tailcut bound. For a cutoff of 12 or 13 bits almost no message can be recovered.

Algorithm 26 Additive Encoding

```
func Encode'($\mu \in \{0,1\}^{\lfloor \frac{n}{u} \rfloor}, u \in \mathbb{N}$)
    for i=0 to $\lfloor \frac{n}{u} \rfloor - 1$ do
        for j=0 to u-1 do
            $\bar{\mathbf{m}}[u \cdot i + j] = \mu[i] \cdot \frac{q-1}{2}$
        end for
    end for
    return $\bar{\mathbf{m}}$
end func
```

Algorithm 27 Additive Decoding

```
func Decode'($\bar{\mathbf{m}} \in \mathcal{R}_q, u \in \mathbb{N}$)
    for i=0 to $\lfloor \frac{n}{u} \rfloor$ do
        $s = 0$
        for j=0 to u-1 do
            $s = s + |\bar{\mathbf{m}}[u \cdot i + j]|$
        end for
        if $s < \frac{u \cdot q}{4}$ then
            $\mu[i] = 0$
        else
            $\mu[i] = 1$
        end if
    end for
    return $\mu$
end func
```

As it turns out the error rate does not significantly increase – even if we remove seven least significant bits of every coefficient and thus have halved the size of $\mathbf{c}_2$. A further option to reduce ciphertext expansion is to omit whole coefficients of $\mathbf{c}_2$ in case they are not used to transfer message bits, e.g., to securely transport a symmetric key. Note that this approach does not affect the concrete security level of the scheme as the modification does not involve any knowledge of the secret key or message and thus does not leak any further information.

Decreasing the Error Rate

A simple approach to deal with decryption errors is to modify the threshold encoding scheme. Instead of encoding one bit into each coefficient of $\mathbf{c}_2$ a plaintext bit can be encoded into u coefficients of $\mathbf{c}_2$. This additive threshold encoding algorithm is shown in Algorithm 26 where Encode' takes as input a plaintext bit-vector μ of length $\lfloor \frac{n}{u} \rfloor$ and outputs the threshold encoded vector $\bar{\mathbf{m}}$ of length n. The decoding procedure Decode', described in Algorithm 27, is given the encoded message vector $\bar{\mathbf{m}}$ affected by an unknown error vector.

The impact on the error rate by using additive threshold encoding ($u = 2$) jointly with the removal of least significant bits is shown in Table 5.2. Note that this significantly lowers the error rate without any expensive encoding or decoding operations and is much more efficient than, e.g., a simple repetition code [MS06]. This method seems especially useful for cases when the message space is not completely used. As an example, when a plain 128-bit AES key has to be transferred it just uses half of the message space of the scheme for $n = 256$.

5.3 High-Performance Implementation

In this section we present our high-speed core implementing RLWEenc, which is using the NTT-aware algorithms as discussed in Section 5.2.1. The goal is to provide a high performance hardware implementation with reasonable area consumption that is also flexible by supporting key generation, encryption, and decryption in one core. Especially the ability to generate keys would enable the usage of our implementation in a key exchange protocol that requires the generation of temporary session keys (ephemeral keys). Our implementation relies on the microcode engine discussed in Chapter 4 and the main additional components required by RLWEenc is a Gaussian sampler, message encoding, and control logic.

5.3.1 High-Speed Gaussian Sampling Based on the CDT

For our implementation we use the cumulative distribution table (CDT) method (sometimes also referred to as inverse transform method) and we refer to works like [Dev86, GPV08, DDLL13b, DG14] and Section 2.5 for more details. When applying this method in general, a table of cumulative probabilities $p_z = \Pr(x \leqslant z : x \leftarrow D_\sigma)$ for integers $z \in [-\tau\sigma, ..., \tau\sigma]$ is computed with a precision of λ bits. For a uniformly random chosen value x from the interval $[0, 1)$ the integer $y \in \mathbb{Z}$ is then returned for which it holds that $p_{z-1} \leq x < p_z$.

In hardware we perform this operation on integers by feeding a uniformly random value into a parallel array of comparators. Each comparator c_i compares its input to the commutative distribution function scaled to the range of the PRNG outputting r bits. As we have to cut the tail at a certain point, we compute the accumulated probability over the positive half (as it is slightly smaller than 0.5) until we reach the maximum value j (e.g., $j = \lceil 2s \rceil$) so that $w = \sum_{k=0}^{j} \rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$. We then compute the values fed into the comparators as $v_k = \frac{2^{r-1}-1}{w}(v_{k-1} + \sum_{k=0}^{j} \rho_\sigma(x)/\rho_\sigma(\mathbb{Z}))$ for $0 < k \leq j$ and with $v_0 = \frac{2^{r-1}-1}{2w}\rho_\sigma(0)/\rho_\sigma(\mathbb{Z})$. Each comparator c_i is preloaded with the rounded value v_i and outputs a one bit if the input was smaller or equal to v_i. A subsequent circuit then identifies the first comparator c_l which returned a one bit and outputs either l or $-l$.

The block diagram of the sampler is shown in Figure 5.1 for the concrete parameter set RLWEenc-Ia $(n{=}256, q{=}7681, s{=}11.32)$ where the output of the sampler is restricted to $[-\lceil 2s \rceil, \lceil 2s \rceil] = [-5.09\sigma, 5.09\sigma]$ and the amount of required randomness is 25 bits per sample. These random bits are supplied by a PRNG for which we used the output of an AES block cipher operating in counter mode. Each 128-bit output block of our AES-based PRNG allows sampling of 5 coefficients. One random bit is used for sign determination while the other 24 bits form a uniformly random value. Finally, the output of the sampler is buffered in a FIFO. When leaving the FIFO, the values are lifted to the target domain $[0, q-1]$. Although it is possible to generate a sampler directly in VHDL by computing the cumulative distribution function on-the-fly during synthesis, we have implemented a Python script to support computation with arbitrary precision.

5.3.2 Design of the Encryption and Decryption Core

For our implementation of RLWEenc we use the medium RLWEenc-Ia $(n = 256, q = 7681, s = 11.31)$ and high security RLWEenc-IIa $(n = 512, q = 12289, s = 12.18)$ parameter sets as proposed in [GFS+12] which are specifically optimized for implementation in hardware (see Table 3.1). To carry out polynomial arithmetic we rely on the microcode engine as discussed in Section 4. Thus the size of the data-path depends mainly on the size of the modulus q and is $\lceil \log_2(q) \rceil$ bits as polynomial coefficients are processed serially in a pipeline. A block diagram of the top-level

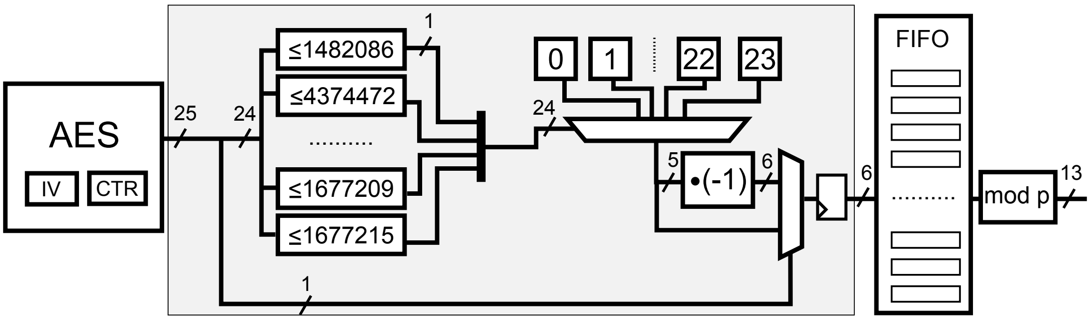

Figure 5.1: Gaussian sampler using the cumulative distribution table (CDT) method and an array of comparators.

module `LWEenc` is given in Figure 5.2. The `LWEenc` module instantiates the microcode engine and uses a block RAM as external interface to export or import ciphertexts $\tilde{\mathbf{c}}_1, \mathbf{c}_2$, keys $\tilde{\mathbf{r}}_2, \tilde{\mathbf{p}}$ or the message μ with straightforward clock domain separation (see again Figure 5.2). The processor is controlled by a finite-state machine (`FSM`) issuing commands to the lattice processor to perform encryption, decryption, key import, or key generation. The microcode engine is configured with three general purpose registers R4-R6 in order to permanently store the public key $\tilde{\mathbf{p}}$, the global constant $\tilde{\mathbf{a}}$ and the private key $\tilde{\mathbf{r}}_2$. More registers for additional key-pairs are also supported, but optional. The implementation supports pre-initialization of registers so that all constant values and keys can be directly included in the bitstream. Note that, for encryption, $\mathbf{c}_1$ and $\mathbf{c}_2$ can be computed independently from the message which is then only added in the last step. This is similar to the mode of operation of a symmetric stream cipher and the XORing of the keystream to the ciphertext.

5.3.3 Results

For performance analysis we primarily focus on Virtex-6 platforms (speed grade -2) but would also like to emphasize that our solution can be efficiently implemented even on a small and low-cost Spartan-6 FPGA. Results were obtained post-place and route (post-PAR) with Xilinx ISE 14.2 and Xilinx ISE 14.7, which was used to re-synthesize the toplevel modules.

Gaussian Sampling

In Table 5.4 we summarize resource requirements of six setups of the implemented comparator-based Gaussian sampler for different tail cuts and precision. The entry *rnd* denotes the number of used random bits to sample one value. Our random number generator is a round based AES in counter mode that computes a 128-bit AES block in 13 cycles and comprises 349 slices, 1,181/350 LUT/FF, two 18K block RAMs and runs with a maximum frequency of about 265 MHz. Combined with this PRNG[2], Gaussian sampling based on the inverse transform method is efficient for small values of s (as typically used for RLWEenc) but would not be suitable for larger Gaussian parameters like, e.g., $s = \sqrt{2\pi}2688 = 6737.8$ for the signature scheme presented in [Lyu12]. While our sampler needs a huge number of random inputs, the AES engine is still able

---

[2] Generation of true random numbers is not in the scope of this work; we refer to the survey by Varchola [Var08] how to achieve this.

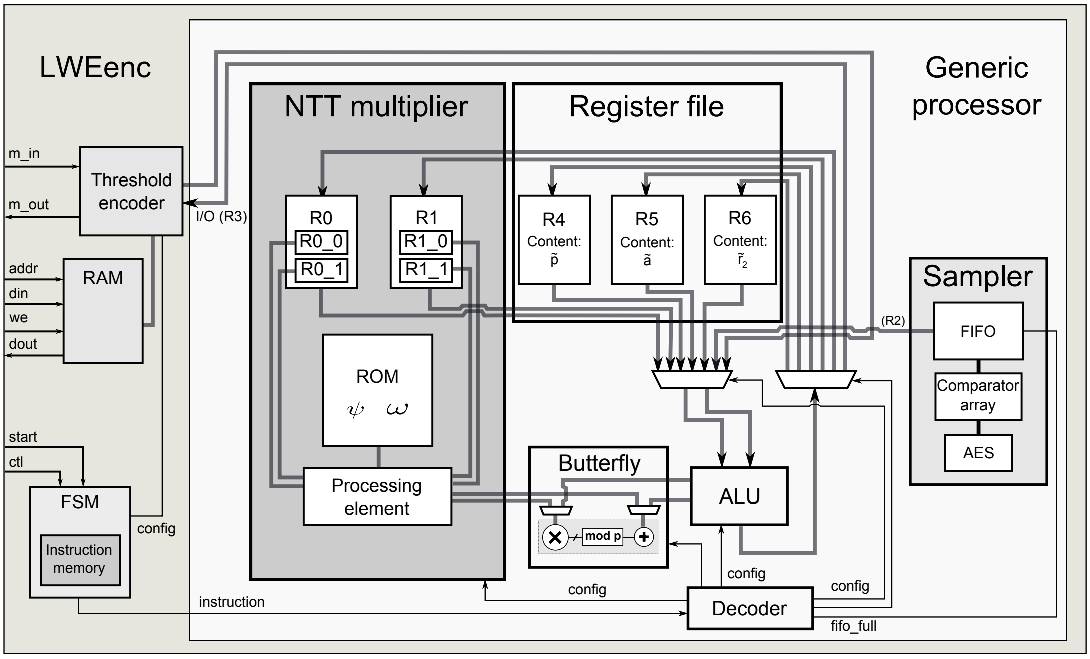

Figure 5.2: Architecture of our RLWEenc core using our microcode engine with three additional registers R4-6.

to generate these numbers (for each encryption we need $3n$ samples). Table 5.4 also shows that it is possible to realize an efficient sampler even for a small statistical distances since its resource consumption of roughly 250 slices is quite moderate (setup III/IV). With additional register levels and pipelining for versions I/II we achieved the overall clock frequency for the whole core reported in Table 5.7 in this section. As the PRNG does not provide enough randomness to sample a value in every clock cycle it is not necessary to evaluate the comparator array in every single cycle so that in particular setups III-VI can use several clock cycles until output is provided. This lowers the critical path and thus allows higher clock frequencies without costs for pipelining registers. Setups V/VI are even more accurate and support (theoretical) requirements of a statistical distance smaller than 2^{-100} [DG14]. However, then a faster PRNG would be required and to encrypt we would need $105 \cdot 3n = 80{,}640$ bits of random input, assuming $n = 256$.

Performance of RLWEenc

Table 5.5 lists the resource consumption and performance of our implementation of RLWEenc. As stated in Section 4.4 our implementation combines key generation, encryption, and decryption in a holistic design and would not significantly benefit from removing any one of these functional units. The only exception might be a decryption-only core in which no Gaussian sampling is needed. It can be seen that our proposed work is very compact as it only uses 11% of the slices of the smallest Virtex-6 and also fits onto the low-cost Spartan-6 LX16 devices (see Table 5.7). The maximum achievable clock frequency of 245 MHz is quite high, especially for a relatively complicated and highly configurable public-key encryption implementation. The

Table 5.4: Performance, resource consumption, and precision of the core part (shaded gray in Figure 5.1) of our Gaussian sampler on a Virtex-6 LX75T (post-PAR).

Setup	s	Max s	rnd	Slices	LUT/FF	MHz
I	11.32	23	25	42	136/5	115
II	12.18	25	25	46	149/5	118
III	11.32	48	85	231	863/6	61
IV	12.18	51	85	255	911/6	61
V	11.32	53	105	314	1157/6	58
VI	12.18	57	105	342	1248/6	50

required time for key generation, encryption, and decryption is well balanced and should meet most real world demands. Additionally we would like to note that our implementation of RLWEenc is fully pipelined and has no data-dependent operations. The processor core does not support any branches and Gaussian sampling based on the inverse transform operates in constant time. Summarizing, all cryptographic operations of our core are timing-invariant. As a consequence, the implementation is protected against side-channel attacks [MOP07] that use timing information of the security algorithm by measuring execution time or cycles.

Table 5.5: Resource consumption and performance of our RLWEenc core on a Virtex-6 LX75T (post-PAR).

	Aspect	Medium security (set Ia) (n=256,q=7681,s=11.32)	High security (set IIa) (n=512,q=12289,s=12.18)
Resources	Slices	1,323	1,601
	LUT/FF	3,644/3,368	4,759/4,484
	18K BRAM	12	13
	DSP48E1	1	1
Performance	MHz	261	245
	$\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{gen}}$ (cycles/time)	7,076/27.11 µs	14,337/58.52 µs
	$\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{enc}}$ (cycles/time)	6,263/24.00 µs	12,750/52.04 µs
	$\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{dec}}$ (cycles/time)	3,200/12.26 µs	6,377/26.03 µs

5.4 Low-Area Implementation

In this section we provide an area efficient implementation of RLWEenc that is still able to achieve high performance. Additionally, we use a result by Brakerski et al. [BLP⁺13]. They show that q is not necessarily required to be prime for security reductions to hold for LWE and cryptanal-

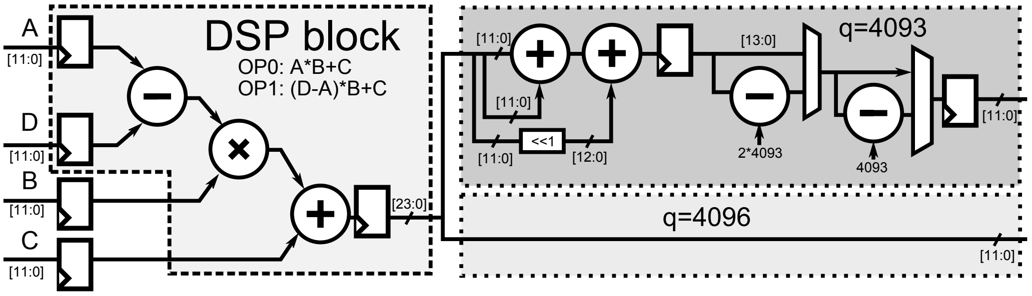

Figure 5.3: Block diagram of our $\log_2(q) \times \log_2(q)$-bit multiplier, $\log_2(q)$ adder, and reduction modulo q for $q = 4093$ and $q = 4096$.

ysis does also not exploit or require that q is a prime[3]. As a power of two modulus allows a significantly more efficient implementation of modular reduction we propose, based on the work of Brakerski et al. [BLP+13], the modified parameter set RLWEenc-Ic $(256, 4096, 8.35)$. In this section we then describe a lightweight implementation of separate encryption and decryption modules for the parameter sets RLWEenc-Ib $(256, 4093, 8.35)$ and RLWEenc-Ic $(256, 4096, 8.35)$ of RLWEenc (see Table 3.1). We are using a pipelined DSP-enabled schoolbook polynomial multiplier and a recently proposed method for efficient sampling from a discrete Gaussian distribution using the Bernoulli distribution and small tables as described in [DDLL13a].

5.4.1 Row-Wise Polynomial Multiplication

For our implementation we used row-wise polynomial multiplication, which can be implemented efficiently with just two counters and a $\log_2(q) \times \log_2(q)$ modular multiplication unit. An advantage over recursive algorithms is the low memory consumption and the immediate modular reduction modulo $\mathbf{x}^n + 1$. In Algorithm 28 we give the $\mathsf{RLWEenc}_{\mathsf{enc}}^{\mathsf{School}}$ algorithm used to compute the encryption operation, which takes the public key $\mathsf{pk} = (\mathbf{a}, \mathbf{p})$ and the binary message vector $\mu \in \{0, 1\}^n$ as inputs. The $\mathsf{SampleGauss}(\tau, \sigma)$ algorithm returns an integer sampled from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$. As we just have one polynomial multiplier we first sample a coefficient of $\mathbf{e}_1$ (step 8), compute a row of $\mathbf{c}_1$ (step 9), and then a row of $\mathbf{c}_2$ (step 13). In every execution of the loop in step 5 we also sample one coefficient of $\mathbf{e}_2$ and $\mathbf{e}_3$. The reason for mixing the sampling into the polynomial multiplication is that we want to minimize buffers in the non-constant time sampler that is used in the low-area implementation (see Section 5.4.2). Sampling the complete $\mathbf{e}_1, \mathbf{e}_2$, or $\mathbf{e}_3$ polynomials at once would require additional storage space or a very fast sampler.

The $\log_2(q) \times \log_2(q)$-bit multiplier, accumulator, and modulo reduction circuit used in step 11 and step 15 is implemented as depicted in Figure 5.3. For the $q = 4096$ case no reduction is necessary as we just need the 12 lowest output bits of the DSP block. For $q = 4093$ we observe that $2^{12} \bmod 4093 = 3$. So we can write $x_{23..0} \bmod 4093 \equiv 2^{12} x_{23..12} + x_{11..0} \equiv 3x_{23..12} + x_{11..0} \equiv (x_{23..12} \ll 1) + x_{23..12} + x_{11..0}$ (by $\ll$ we denote a shift-left operation). This result can then be

---

[3] In recent work like [EHL14, ELOS15] several restrictions regarding the choice of parameters for LWE and RLWE instances have been shown. However, when using a power of two dimension n and a power of two modulus q the attacks do not appear to be successful.

Algorithm 28 RLWEenc Encryption Using the Schoolbook Algorithm

```
func $\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{enc}}(\mathsf{pk}{=}(\mathbf{a},\mathbf{p}), \mu{\in}\{0,1\}^n)$
  for $i = 0$ do $n-1$
    $\mathbf{c}_1[i] \leftarrow 0$, $\mathbf{c}_2[i] \leftarrow 0$
  end for
  for $i = 0$ do $n-1$
    $\mathbf{c}_1[i] \leftarrow \mathbf{c}_1[i] + \mathsf{SampleGauss}(\tau, \sigma)$
    $\mathbf{c}_2[i] \leftarrow \mathbf{c}_2[i] + \mathsf{SampleGauss}(\tau, \sigma)$
    $e \leftarrow \mathsf{SampleGauss}(\tau, \sigma)$
    for $j = 0$ do $n-1$
      $h \leftarrow i + j \bmod n$
      $\mathbf{c}_1[h] \leftarrow (\mathbf{c}_1[h] + (-1)^{\lfloor \frac{i+j}{n} \rfloor}\mathbf{a}[j]e) \bmod q$
    end for
    for $j = 0$ do $n-1$
      $h \leftarrow i + j \bmod n$
      $\mathbf{c}_2[h] \leftarrow (\mathbf{c}_2[h] + (-1)^{\lfloor \frac{i+j}{n} \rfloor}\mathbf{p}[j]e) \bmod q$
    end for
  end for
  for $i = 0$ do $n-1$
    $\mathbf{c}_2[i] \leftarrow (\mathbf{c}_2[i] + \lfloor \frac{q}{2} m_i \rceil) \bmod q$
  end for
  return $\mathbf{c}_1, \mathbf{c}_2$
end func
```

reduced into the range $[0, 4092]$ by at maximum two subtractions of q. The implemented circuit is shown in Figure 5.3 and translates efficiently into hardware. Note that we are using a full $\log_2(q) \times \log_2(q)$-bit multiplier, although the values sampled from D_σ are small and only in the range $[-\tau\sigma, \tau\sigma]$ (for the specific parameter set we set $\tau = 12$). In general this would allow a smaller signed multiplier. However, in this case the reduction of both positive and negative multiplication results would be more complicated. Additionally, we already use a DSP block (see [Xil09a]) which natively supports a 18×18-bit multiplication without additional resource usage. A multiplication by a negative number is implemented by choosing the corresponding mode of operation of the DSP block. The mode $AB + C$ is used for unsigned multiplication and the mode $(D - A)B + C = (q - A)B + C$ for signed multiplication, respectively. As this mode is supported directly by the DSP block it does not require further resources, e.g., a multiplexer or subtraction circuit.

In Figure 5.4 we show the block diagram of the encryption core. We use one 9 Kb dual-port block RAM (BRAM9) to store the public key $\mathbf{a}, \mathbf{p}$ $(2 \cdot 256 \cdot 12 = 6144$ bits) and one dual-port BRAM9 to hold temporary variables and the final ciphertext $\mathbf{c}_1, \mathbf{c}_2$ $(2 \cdot 256 \cdot 12 = 6144$ bits). The multiplication is basically controlled by two counters implementing the interleaved multiplication where the state machine can select whether $(\mathbf{c}_1, \mathbf{a})$ or $(\mathbf{c}_2, \mathbf{p})$ are accessible (`set_h`). A block diagram of the modular multiplication design is given in Figure 5.3. To save a block RAM we

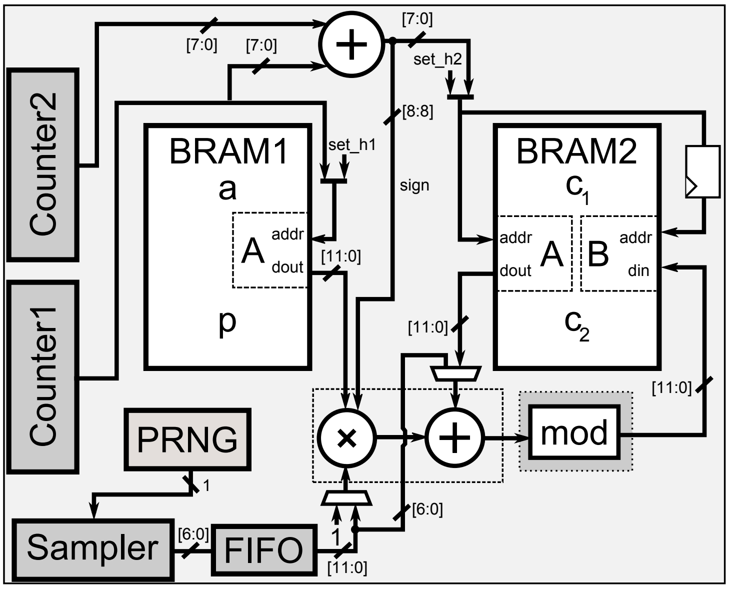

Figure 5.4: Block diagram of the lightweight RLWEenc encryption circuit where the public key $\mathbf{a}, \mathbf{p}$ is stored in `BRAM1` and the ciphertext $\mathbf{c}_1, \mathbf{c}_2$ in `BRAM2`.

store the private key in a 256×7-bit ROM, which is realized using memory LUTs. In case a key update is necessary it is also possible to use a RAM with low overhead. As the decryption circuit requests the ciphertext coefficient by coefficient, only one dual-port BRAM9 is necessary.

5.4.2 Area Efficient Rejection Sampling

Implementing discrete Gaussian sampling with high precision and low resource consumption is challenging. Simple rejection sampling using floating-point arithmetic is definitely too costly, especially on an FPGA without hardware supported floating point operations. For the cumulative distribution table (CDT) approach a precomputed table of roughly $\tau\lambda\sigma$ bits is required. While the inversion/CDT method using comparators has shown high performance (see Section 5.3.1) it also leads to a rather large implementation. Roy et al. [RVV13] provided an implementation of a Gaussian sampler using the Knuth-Yao method which allows halving the size of the precomputed table. In order to further reduce the need for precomputation, we consider a proposal by Ducas et al. [DDLL13a]. They propose a simple method to sample according to the Bernoulli distribution[4] $\mathcal{B}_{\exp(-x/f)}$ with very low memory overhead using $\log_2(\lceil\max(x)\rceil)$ precomputed entries with λ-bit precision (see Algorithm 29). The sampling with constant and precomputed biases in step 5 can be performed by sampling the binary expansion of a uniform number $r \in [0,1)$ with λ bits of precision and returning one if and only if $r < c$. To save random bits we evaluate the comparison bit-by-bit instead of sampling a λ-bit r at once.

---

[4] The Bernoulli distribution $\mathcal{B}_c$ outputs one (acceptance) with probability $c \in [0,1)$ and zero (rejection) otherwise.

Algorithm 29 Bernoulli Sampling: $\mathcal{B}_{\exp(-x/f)}$

```
**Precomputation:** $c_i = \exp(-2^i/f)$ for $0 \le i \le l-1$
func SampleBer($x \in [0, 2^l)$)
  for $i = l-1$ do 0
    if $x_i = 1$ then
      sample $A_i \leftarrow \mathcal{B}_{c_i}$
    end if
    if $A_i = 0$ then
      return 0
    end if
  end for
  return 1
end func
```

Algorithm 30 Rejection Sampling Using Algorithm 29

```
func SampleGauss($\tau, \sigma$)
  $u \leftarrow$ Uniform($0, \lceil \tau\sigma \rceil$)
  if $\mathcal{B}_{\exp(-u^2/(2\sigma^2))} = 0$ then
    restart
  end if
  if $u = 0$ then
    $b \leftarrow$ Uniform($0, 1$)
  end if
  if $b = 0$ then
    restart
  end if
  $b \leftarrow$ Uniform($0, 1$)
  return $(-1)^b u$
end func
```

The general idea of rejection sampling is to choose a uniformly random $u \in \{-\tau\sigma, ..., \tau\sigma\}$ which is then accepted with a probability proportional to $\exp(-x^2/2\sigma^2)$. In Algorithm 30 we provide a sampling procedure which picks a uniformly random $u \in \{0, ..., \tau\sigma\}$ and uses Algorithm 29 to accept proportional to $\exp(-u^2/2\sigma^2)$. In order to sample in the range $\{-\tau\sigma, ..., \tau\sigma\}$ we then reject the output zero ($u = 0$) with probability $\frac{1}{2}$ and sample a sign bit. The number of required table entries is $\log_2(\tau^2\sigma^2)$ and for $\sigma = 3.33$, $\tau = 12$ and $\lambda = 80$ we get a table size of 880 bits implemented in memory LUTs (LUTM). Although we have eliminated the need for high precision evaluation of the exp() function, we still need $2\tau/\sqrt{2\pi} \approx 10$ trials and thus a high number of uniformly random bits. This is mitigated by using an implementation of the resource efficient Trivium stream cipher as a PRNG that can generate a sufficient amount of randomness to match the speed of the polynomial arithmetic. To ensure a constant run-time of the encryption we have implemented a small buffer FIFO (see Figure 5.4) which holds up to

16 sampled values. They are saved in the range $[-\lceil\sigma\tau\rceil, \lceil\sigma\tau\rceil]$ and then lifted into the range $[0, q-1]$ at the output port of the FIFO.

5.4.3 Results

Performance results and resource consumption of our low area implementation of $\mathsf{RLWEenc}$ using the $\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{enc}}$ and $\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{dec}}$ algorithms are provided in Table 5.6. Our results were obtained post-PAR with Xilinx ISE 14.6 and a small Spartan-6 LX9 (speed grade -2) as target device. Synthesis and place-and-route options were optimized for small area and we also utilize the memory LUT capabilities of the device (LUTM). In Table I we give the overall resource consumption of the encryption and decryption cores and demonstrate the significant advantage in applying the result of Brakerski et al. [BLP+13] to the design. This is especially striking for the decryption core since the core for $q = 4096$ needs only 63% of the slices of the core for $q = 4093$. The size of the encryption core is dominated by the resource consumption of the sampler. Still the core for $q = 4096$ is 19 slices smaller than the core for $q = 4093$. The achieved clock frequencies of 128/144 MHz and 179/189 MHz match the requirements of typical lightweight scenarios and result in 934/1,057 and 2,700/2,849 encryption and decryption operations per second, respectively (each handling n bits of plaintext).

Table 5.6: Resource consumption and performance of our area optimized FPGA implementation of $\mathsf{RLWEenc}$.

Operation	Parameter	LUT/LUTM/ FF/Slice	BRAM9/ DSP	MHz	Cycles	OP/s
$\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{enc}}$	$(256, 4093, 8.35)$	360/36/290/114	2/1	128	avg. 136,986	934
$\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{enc}}$	$(256, 4096, 8.35)$	282/35/238/95	2/1	144	avg. 136,212	1,057
$\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{dec}}$	$(256, 4093, 8.35)$	162/18/136/51	1/1	179	avg. 66,304	2,700
$\mathsf{RLWEenc}^{\mathsf{School}}_{\mathsf{dec}}$	$(256, 4096, 8.35)$	94/18/87/32	1/1	189	avg. 66,338	2,849

5.5 Comparison with Related Work

In this section we compare our high performance and low area core with related work. In Table 5.7 we provide our results as well as results by Göttert et al. [GFS+12], Roy et al. [RVM+14], our original publication [PG13], and we list other relevant asymmetric schemes. We also add performance figures for a Spartan-6 instantiation.

Note that a detailed comparison with [GFS+12] is not possible due to inaccuracies of synthesis results[5]. Figures for clock frequency, overall slice consumption, and cycles counts for individual operations or the whole encryption block are not given in [GFS+12]. We therefore can only refer to numbers providing the resource consumption of registers and LUT usage. For a rough comparison we apply the throughput to area (T/A) metric and define area equivalent to the

---

[5] The Virtex-6 LX240T FPGA used in [GFS+12] was overmapped so that the subsequent PAR step providing final results could not have been performed.

usage of LUTs due to the restriction mentioned above. It turns out that our implementation for $n = 256$ is 40 times smaller regarding key generation, 81 times smaller for encryption and 34 times smaller for decryption, at a loss of a factor of about 3 and 1.5 in performance for encryption and decryption, respectively. When employing a $\frac{\text{bit/s}}{\text{LUT}}$ metric[6] for medium security encryption we achieve $\frac{10.66 \cdot 10^6 \text{ bits}}{4549 \text{LUTs}} = 2{,}927$ while the work presented in [GFS+12] gives $\frac{31.8 \cdot 10^6 \text{ bits}}{298{,}016 \text{ LUTs}} = 106$. This results in an improvement of a factor of roughly 27.6.

The implementation by Roy et al. [RVM+14] appeared after the original publication [PG13] of the work presented in this chapter and is significantly smaller and also faster. Better efficiency is gained through the use of a more efficient microcode engine and by an implementation of a Knuth-Yao table-based discrete Gaussian sampler. For a comparison of our microcode engine with Roy et al. [RVM+14] we refer to Section 4.6.3. The main improvement in this regard appears to be that the polynomial multiplier of Roy et al. does not have separate NTT-enabled registers but a large address space which can be directly accessed by the multiplier. Thus no loading of polynomials into the NTT-enabled registers and block memory space is required.

In comparison with a recent implementation of the code-based Niederreiter scheme [HG12] our decryption is faster and we also use fewer resources on the same platform. Another natural target for comparison is the patent-protected NTRU scheme which has been implemented on a large number of architectures [BCE+01, ABF+08, HVP10]. The implementation in [KY09] is clearly faster than ours. However, the implemented NTRU(251,3,12) variant in [KY09] seems to be less secure than RLWEenc-Ia [HHHW09]. Unfortunately, we are not aware of any newer NTRU FPGA implementations in order to determine the impact of increased security parameters on runtime and area consumption. In software, NTRU even seems to be rather slow for higher security levels what can be obtained from the 256-bit secure NTRU software implementation (`ntruees787ep1`) benchmarked using the eBACS framework [BL] with secret/public key sizes of 1,854/1,574 bytes and a ciphertext of 1,574 bytes. For the ideal lattice-based NTRU version presented in [SS11], no hardware implementation has been published yet. In comparison with ECC over prime curves (i.e., a single point multiplication [GP08]) and RSA (random-exponent 1024-bit exponentiation [Suz07]) our implementation is by an order of magnitude faster, scales better for higher security levels, and also consumes less resources. However, we are not able to beat the recent binary curve implementation of Rebeiro et al. [RRM12] in terms of throughput and performance.

5.6 Conclusion and Future Work

In this chapter we have shown that the RLWEenc scheme can be efficiently implemented on reconfigurable hardware for high-performance and low-area application scenarios. Our results for the high-performance implementation were even further improved in related work by Roy et al. [RVM+14]. However, we still see some opportunities for further work. While the Knuth-Yao sampler employed in [RVM+14] is very efficient, it might also be possible to tune the CDT approach for high speed by using techniques from Section 2.5.4. Additionally, RLWEenc would directly benefit from a more efficient implementation of the microcode engine and a more efficient NTT-based polynomial multiplier.

---

[6] For this comparison we assumed that for each encryption 256 bits are transmitted.

Table 5.7: Performance comparison of our implementations with other implementations of 80-bit to 128-bit secure PKEs.

Scheme	Device	Resources (LUT/FF/BRAM/DSP)	OP	Speed
RLWEenc (speed, **our work**) $(n = 256, q = 7681)$	S6LX16 @159 MHz	3,564/3,307/13/1 - -	Gen Enc Dec	44.50 μs 39.39 μs 20.13 μs
RLWEenc (speed, **our work**) $(n = 256, q = 7681)$	V6LX75T @261 MHz	3,644/3,368/12/1 - -	Gen Enc Dec	27.11 μs 24.00 μs 12.26 μs
RLWEenc(area, **our work**) $(n = 256, q = 7681)$	S6LX9 S6LX9	396/290/2/1 180/136/1/1	Enc Dec	1.07 ms 370.00 μs
RLWEenc [PG13] $(n = 256, q = 7681)$	S6LX16 @160 MHz	4121/3513/14/1 - -	Gen Enc Dec	45.22 μs 42.88 μs 27.51 μs
RLWEenc [PG13] $(n = 256, q = 7681)$	V6LX75T @262 MHz	4549/3624/12/1 - -	Gen Enc Dec	27.61 μs 26.19 μs 16.80 μs
RLWEenc [RVM+14] $(n = 256, q = 7681)$	V6LX75T V6LX75T	1,349/860/2/1 -	Enc Dec	20.10 μs 9.10 μs
RLWEenc [GFS+12] $(n = 256, q = 7681)$	V6LX240T V6LX240T V6LX240T	146,718/82,463/-/- 298,016/143,396/-/- 124,158/65,174/-/-	Gen Enc Dec	- 8.05 μs 8.10 μs
Niederreiter [HG12] (80-bit security)	V6LX240T V6LX240T	888/875/17/- 9,409/12,861/9/-	Enc Dec	0.66 μs 57.78 μs
QC-MDPC [vMG14] (80-bit security)	V6LX240T V6LX240T	224/120/1/- 568/412/3/-	Enc Dec	2.2 ms 13.4 ms
NTRU [KY09]	XCV1600E	27,292/5,160 -	Enc Dec	1.54 μs 1.41 μs
1024-bit mod. Exp. [Suz07]	XC4VFX12	3,937 slice/17 DSP	-	1.71 ms
ECC-P224 [GP08]	XC4VFX12	1,825/1,892/11/26	-	365.10 μs
ECC-B233 [RRM12]	XC5VLX85T	18,097 LUT/5,644 slice	-	12.30 μs

Another area of future work are high-level protocols. Currently, the RLWEenc scheme is mainly used for implementations of lattice-based public-key encryption. However, often key exchange and key transport is an important application and public-key encryption is just a building block in a protocol. It would be interesting to investigate how efficient key exchange algorithms like the one presented in [BCNS15, ZZD+15] are. Moreover, it appears possible to exploit synergies between a public-key encryption and a signature core with regard to polynomial multiplication (RLWEenc and BLISS can be instantiated with the same dimension n and modulus q). Such a core could be used for authenticated key exchange (AKE) and it is an open question if such a design would be more efficient than a direct implementation of a dedicated AKE protocol like the one given in [ZZD+15].

Chapter 6

Implementation of Ring-LWE Encryption on an 8-bit Microcontroller

In this chapter we study the efficiency of RLWEenc *public-key encryption on an 8-bit Atmel ATxmega128 microcontroller and implement the number theoretic transform (NTT) as well as schoolbook multiplication and Karatsuba's algorithm. We specifically focus on tuning the NTT and provide an approach that allows to significantly lower the runtime of polynomial multiplication and thus encryption and decryption in* RLWEenc*. Our final implementation that uses the NTT is faster than previous work and achieves a small memory footprint. This chapter is an extended version of [POG15a, POG15b].*

Contents of this Chapter

6.1 Introduction

Besides the previously covered threat of quantum computers, it has been shown that RSA and ECC are quite inefficient on very small and constrained devices like 8-bit AVR microcontrollers [GPW^{+}04, HS13]. Thus asymmetric cryptosystems based on hard problems on ideal lattices might be a viable alternative to established schemes on these platforms. A particular advantage of RLWEenc, besides short keys, applicability of the NTT, and simplicity (see Chapter 3.4), is that the computations are performed on polynomials with small coefficients. Many operations on small coefficients of size less than 16 bits can presumably be easier processed on an 8-bit architecture than the multiprecision arithmetic required for RSA and ECC. However, so far only few works like [BSJ14, BJ14] cover the implementation of lattice-based cryptography on constrained 8-bit architectures.

It is also worth mentioning that current works dealing with high performance implementation of ideal lattice-based cryptography usually rely on the straightforward Cooley-Tukey (CT) radix-2 decimation-in-time (DIT) NTT algorithm (e.g., [PDG14a, BSJ14, RVM+14, dCRVV15]). However, by taking a closer look at works on the implementation of the highly related FFT [CG00, CP01], it becomes evident that the sole focus on CT radix-2 DIT algorithms prevents further optimizations of the NTT, especially given the constraints of an 8-bit architecture.

Furthermore, implementations on constrained devices should not be solely written for performance as RAM consumption and code size can also be important for certain applications. An example scenario is a smart card which might only require few asymmetric encryption operations per second. But in this case large code-size and RAM consumption might significantly contribute to the costs of the whole device. Thus, it makes sense to investigate algorithms with worse asymptotic runtime than the NTT that could lead to a more compact implementation. Additionally, similar to the low-area implementation in Section 5.4, not using the NTT allows more freedom when selecting parameters, e.g., usage of power-of-two moduli, and thus performance and code saving due to the lack of modular reduction routines.

6.1.1 Related Work

The RLWEenc public-key encryption scheme [LPR10b, LPR10c, LP11] is covered in Section 3.4. In Chapter 5 we discuss decryption errors and two hardware designs optimized for performance and area, respectively. Implementations of RLWEenc on the 8-bit ATxmega and ATmega family of microcontrollers have been proposed in [BSJ14, BJ14] and an implementation on a Cortex-M4F device is given in [dCRVV15]. Details on the (software) implementation of the FFT and NTT can be found in works like [CG00, CP01, Har14, MBFK14]. A vectorized implementation of the NTT on standard Intel/AMD CPUs targeting lattice-based cryptography is shown in [GOPS13]. Optimizations of the NTT computation for lattice-based cryptography, with a focus on hardware implementation, have been proposed in [APS13, RVM+14].

6.1.2 Contribution

The main contribution of this chapter is the optimization of the NTT and its fast implementation. To optimize the NTT we review different approaches and varieties of FFT algorithms and then adapt these algorithms for the polynomial multiplication use-case prevalent in ideal lattice-based cryptography. Improvements compared to previous work are mainly achieved by merging certain operations into the NTT itself (multiplication by n^{-1}, powers of ψ and ψ^{-1}) and by removing the expensive bit-reversal step. Additionally, we provide an efficient implementation of these NTT algorithms on the 8-bit AVR/ATxmega architecture. Our work shows that lattice-based public-key encryption is fast on the AVR architecture and an encryption operation takes only 27 ms while 6.7 ms are required for decryption. Additionally, we investigate fast polynomial multiplication using the schoolbook method and Karatsuba's algorithm. However, while both algorithms allow a slightly smaller code size they are also much slower compared to the NTT.

6.2 Faster NTTs for Lattice-Based Cryptography

In this section we examine fast algorithms for the computation of the NTT and show techniques to speed up polynomial multiplication for lattice-based cryptography[1]. The most straightforward implementation of the NTT is a Cooley-Tukey radix-2 decimation-in-time (DIT) approach [CT65] that requires a bit-reversal step as the algorithm takes bit-reversed input and produces naturally ordered output (from now on referred to as $\mathrm{NTT}^{CT}_{bo\rightarrow no}$). To compute the NTT as defined in Section 2.4.2 the $\mathrm{NTT}^{CT}_{bo\rightarrow no}$ algorithm applies the Cooley-Tukey (CT) butterfly, which computes $a' \leftarrow a + \omega b$ and $b' \leftarrow a - \omega b$ for some values of $\omega, a, b \in \mathbb{Z}_q$, overall $\frac{n \log_2(n)}{2}$ times. The biggest disadvantage of relying solely on the $\mathrm{NTT}^{CT}_{bo\rightarrow no}$ algorithm is the need for bit-reversal, multiplication by constants, and that it is impossible to merge the final multiplication by powers of ψ^{-1} into the twiddle factors (powers of ω) of the inverse NTT (see [RVM+14]). With the assumption that twiddle factors are stored in a table and thus not computed on-the-fly it is possible to further simplify the computation and to remove bit-reversal and to merge certain steps. This assumption makes sense on constrained devices like the ATxmega, which have a rather large read-only flash.

6.2.1 Merging the Inverse NTT and Multiplication by Powers of ψ^{-1}

In [RVM+14] Roy et al. use the standard $\mathrm{NTT}^{CT}_{bo\rightarrow no}$ algorithm for a hardware implementation and show how to merge the multiplication by powers of ψ (see Section 2.4.2) into the twiddle factors of the forward transformation. However, this approach does not work for the inverse transformation due to the way the computations are performed in the CT butterfly as the multiplication is carried out before the addition. In this section we show that it is possible to merge the multiplication by powers of ψ^{-1} during the inverse transformation using a fast decimation-in-frequency (DIF) algorithm [GS66]. The DIF NTT algorithm splits the computation into a sub-problem on the even outputs and a sub-problem on the odd outputs of the NTT and has the same complexity as the $\mathrm{NTT}^{CT}_{bo\rightarrow no}$ algorithm. It requires usage of the so-called Gentleman-Sande (GS) butterfly which computes $a' \leftarrow a + b$ and $b' \leftarrow (a - b)\omega$ for some values of $\omega, a, b \in \mathbb{Z}_q$. Following [CG00, Section 3.2], where ω_n is an n-th primitive root of unity and by ignoring the multiplication by the scalar n^{-1}, the inverse NTT and application of PowMul_ψ can be defined as

$$\mathbf{a}[r] = \psi^{-r} \sum_{\ell=0}^{n-1} \mathbf{A}[\ell]\omega_n^{-r\ell} = \psi^{-r} \left(\sum_{\ell=0}^{\frac{n}{2}-1} \mathbf{A}[\ell]\omega_n^{-r\ell} + \sum_{\ell=0}^{\frac{n}{2}-1} \mathbf{A}[\ell + \frac{n}{2}]\omega_n^{-r(\ell+\frac{n}{2})} \right) \tag{6.1}$$

$$= \psi^{-r} \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] + \mathbf{A}[\ell + \frac{n}{2}]\omega_n^{-r\frac{n}{2}} \right) \omega_n^{-r\ell}, r = 0, 1, \ldots, n-1. \tag{6.2}$$

---

[1] Most of the techniques discussed in this section have already been proposed in the context of the fast Fourier transform (FFT). However, they have not yet been considered to speed up ideal lattice-based cryptography (at least not in works like [RVM+14, dCRVV15, PDG14a, BSJ14]). Moreover, some optimizations and techniques are mutually exclusive and a careful selection and balancing has to be made.

When r is even this results in

$$\mathbf{a}[2k] = \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] + \mathbf{A}[\ell + \frac{n}{2}]\right) \omega_{\frac{n}{2}}^{-k\ell} (\psi^2)^{-k} \tag{6.3}$$

and for odd r in

$$\mathbf{a}[2k+1] = \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] - \mathbf{A}[\ell + \frac{n}{2}]\omega_n^{-\ell}\right) \omega_{\frac{n}{2}}^{-k\ell} \psi^{-(2k+1)} \tag{6.4}$$

$$= \sum_{\ell=0}^{\frac{n}{2}-1} \left(\left(\mathbf{A}[\ell] - \mathbf{A}[\ell + \frac{n}{2}]\right) \psi^{-1}\omega_n^{-\ell}\right) \omega_{\frac{n}{2}}^{-k\ell} (\psi^2)^{-k}, k = 0, 1, \ldots, \frac{n}{2} - 1. \tag{6.5}$$

The two new half-size sub-problems where ψ is exchanged by ψ^2 can now be again solved using the recursion. As a consequence, when using an in-place radix-2 DIF algorithm it is necessary to multiply all twiddle factors in the first stage by ψ^{-1}, all twiddle factors in the second stage by ψ^{-2} and in general by ψ^{-2^s} for stage $\{0, 1, \ldots, \log_2(n) - 1\}$ to merge the multiplication by powers of ψ^{-1} into the inverse NTT (see Figure 6.1 for an illustration). In case the PowMul_{ψ} or $\mathsf{PowMul}_{\psi^{-1}}$ operation is merged into the NTT computation we denote this by an additional superscript ψ or ψ^{-1}, e.g., as $\mathrm{NTT}_{bo \to no}^{CT,\psi}$.

6.2.2 Removing Bit-Reversal

For memory efficient and in-place computation a reordering or so-called bit-reversal step is usually applied before or after an NTT/FFT transformation due to the required reversed input ordering of the $\mathrm{NTT}_{bo \to no}^{CT}$ algorithm used in works like [RVM+14, PDG14a, BSJ14, dCRVV15].

However, by manipulation of the standard iterative algorithms and independently of the used butterfly (CT or GS) it is possible to derive natural order to bit-reversed order ($no \to bo$) as well as bit-reversed to natural order ($bo \to no$) forward and inverse algorithms. The derivation of FFT algorithms with a desired ordering of inputs and outputs is described in [CG00] and we follow this description to derive the NTT algorithms $\mathrm{NTT}_{bo \to no}^{CT}$, $\mathrm{NTT}_{no \to bo}^{CT}$, $\mathrm{NTT}_{no \to bo}^{GS}$, and $\mathrm{NTT}_{bo \to no}^{GS}$, as well as their respective inverse counterparts. It is also possible to construct self-sorting NTTs ($no \to no$) but in this case the structure becomes irregular and temporary memory is required (see [CG00]).

6.2.3 Combination of Optimization Techniques

The optimizations discussed in this section so far can be used to generically optimize polynomial multiplication in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$. However, for lattice-based cryptography there are special conditions that hold for most practical algorithms; in the NTT-enabled algorithms of RLWEenc and BLISS every point-wise multiplication (denoted by $\circ$) is performed with a constant and a variable, usually a randomly sampled polynomial. Thus the most common operation in lattice-based cryptography is *not* simple polynomial multiplication but multiplication of a (usually random) polynomial by a constant polynomial (i.e., global constant or public key). Thus, the scaling factor n^{-1} can be multiplied into the pre-computed and pre-transformed constant $\tilde{\mathbf{a}} =$

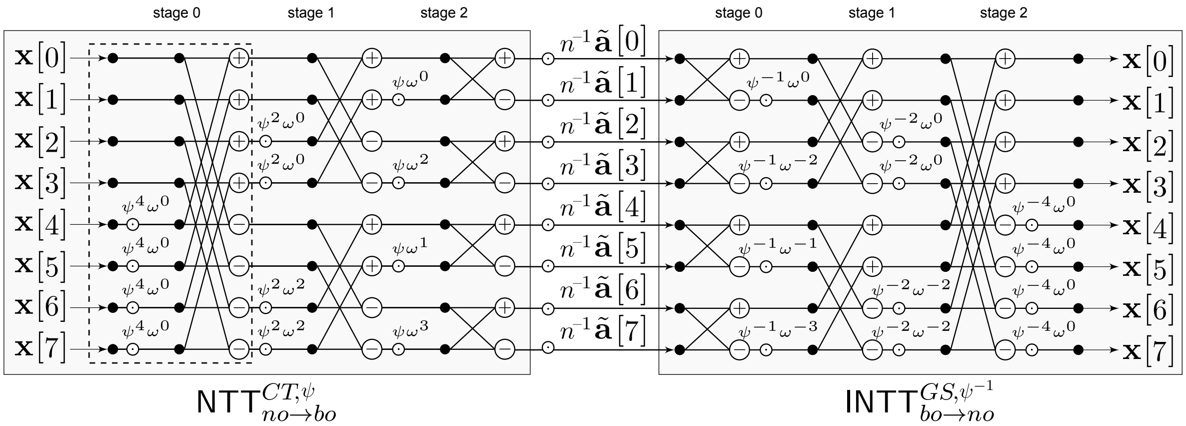

Figure 6.1: Signal flow graph for a multiplication of a polynomial $\mathbf{x}$ by a pre-transformed polynomial $\tilde{\mathbf{a}} = \mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{a})$, using the $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ algorithms.

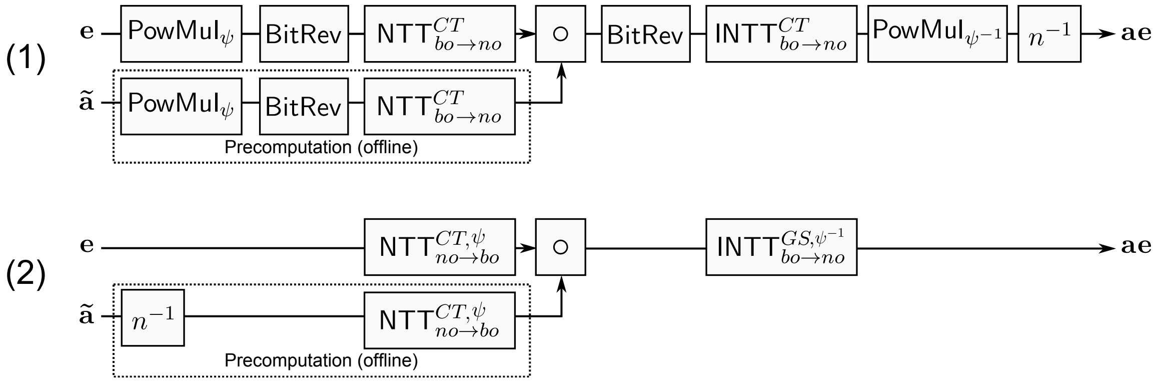

Figure 6.2: Comparison of our NTT implementation using $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ with a naive implementation of polynomial multiplication using the straightforward NTT.

$n^{-1}\mathrm{NTT}(\mathbf{a})$. Taking into account that we also want to remove the need for bit-reversal and want to merge the multiplication by powers of ψ into the forward and inverse transformation (as discussed in Section 6.2.1) we propose to use an $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ for the forward transformation and an $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ for the inverse transformation. In this case a polynomial multiplication $\mathbf{c} = \mathbf{a}\cdot\mathbf{e}$ can be implemented without bit-reversal as $\mathbf{c} = \mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}\left(\mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{a})\circ\mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{e})\right)$. In Figure 6.2 we compare the necessary blocks for the straightforward approach and our proposal and provide an example flow diagram for $n = 8$ in Figure 6.1. For more details, pseudo-code of $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ is provided in Algorithm 31 and pseudo-code of $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ is given in Algorithm 32.

Algorithm 31 CT Forward NTT

```
Precondition: Store $\psi^i$ for $i = 0 \dots n-1$ in bit-reversed order in $psi^*$
func $\mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{a})$
    $m \leftarrow 1$
    $k \leftarrow n/2$
    while $m < n$ do
        for $i = 0$ to $m - 1$ do
            $jFirst \leftarrow 2 \cdot i \cdot k$
            $jLast \leftarrow jFirst + k - 1$
            $\psi_i \leftarrow psi^*[m + i]$
            for $j = jFirst$ to $jLast$ do
                $l \leftarrow j + k$
                $t \leftarrow \mathbf{a}[j]$
                $u \leftarrow \mathbf{a}[l] \cdot \psi_i$
                $\mathbf{a}[j] \leftarrow t + u \mod q$
                $\mathbf{a}[l] \leftarrow t - u \mod q$
            end for
        end for
        $m \leftarrow m \cdot 2$
        $k \leftarrow k/2$
    end while
    return $\mathbf{a}$
end func
```

Algorithm 32 GS Inverse NTT

```
Precondition: Store $\psi^{-i}$ for $i = 0 \dots n-1$ in bit-reversed order in $invpsi^*$
func $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}(\mathbf{a})$
    $m \leftarrow n/2$
    $k \leftarrow 1$
    while $m > 1$ do
        for $i = 0$ to $m - 1$ do
            $jFirst \leftarrow 2 \cdot i \cdot k$
            $jLast \leftarrow jFirst + k - 1$
            $\psi_i \leftarrow invpsi^*[m + i]$
            for $j = jFirst$ to $jLast$ do
                $l \leftarrow j + k$
                $t \leftarrow \mathbf{a}[j]$
                $u \leftarrow \mathbf{a}[l]$
                $\mathbf{a}[j] \leftarrow c + d \mod q$
                $\mathbf{a}[l] \leftarrow (c - d) \cdot \psi_i \mod q$
            end for
        end for
        $m \leftarrow m/2$
        $k \leftarrow k \cdot 2$
    end while
    return $\mathbf{a}$
end func
```

6.3 Implementation of RLWEenc Using the NTT

For the use in RLWEenc and BLISS we focus on the optimization of the $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformations. We implemented both algorithms in C and optimized modular multiplication using assembly language.

6.3.1 Implementation of the NTT

For the use in RLWEenc we focus on the optimization of the $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformations.

Modular Multiplication

To implement the NTT according to Section 6.2, a DIT $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and a DIF $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformation are required and the most expensive computation in both algorithms is the computation of approximately $\frac{n \log_2(n)}{2}$ butterflys consisting of integer multiplication and reduction modulo q. In [BSJ14] Boorghany, Sarmadi, and Jalili report that most of the runtime of their FFT/NTT is spent on the computation of modulo operations. They review modular reduction

```
mm12289(a, b)
uint8_t v[4];
c = a*b;
v = 12289 << 14

REDUCE28:
if (c >= v)
        c = c -v
v = v << 1;

REDUCE27:
if (c >= v)
        c = c -v
v = v << 1;
REDUCE26:
if (c >= v)
        c = c -v
v = v << 1;

REDUCE25:
if (c >= v)
        c = c -v
v = v << 1;
REDUCE24:
if (c >= v)
        c = c -v
v = v << 1;

...
```

Figure 6.3: Modular multiplication for $q = 12289$.

algorithms for suitability and propose an approximate variant of Barrett reduction [Bar86] that leads to an FFT/NTT that is 1.26 times faster for $n = 256$ than one using the compiler generated modulo reduction (see Table 4 of [BSJ14]). However, a straightforward implementation using the C %-operator with a constant modulus is quite expensive and requires around 600 cycles in our own experiments due to the generic libc modular reduction (`call __udivmodsi4`). As a consequence, the software of the authors of [BSJ14] still consumes approx. $\frac{754668}{\frac{256}{2}\log_2(256)} = 737$ cycles for one FFT/NTT butterfly.

Another approach is a subtract-and-shift algorithm which loads the shifted modulus as constant and the input into a temporary register. It then continues to compare the value in the temporary register to this modulus, subtracts if the input is larger or equal to the modulus and then shifts the modules by one to the right. This continues until the shifted modulus is equal to the original modulus. The pseudo-code for this operation is shown in Figure 6.3. The biggest improvement in assembly stems from the ability to limit the operations on the active registers. As an example, when the input is 28-bits wide the first comparisons and shifts have to be performed on four registers (`REDUCE28` to `REDUCE25`), but after four iterations all operations (comparison, subtraction, shift) have to be performed only on three registers (from `REDUCE24`), or two registers (from `REDUCE16`). This approach guarantees that one modular multiplication for $q = 7681$ takes *at most* 216 cycles. While we do not take into account constant time operation for side-channel protection, the implementation can be made trivially constant time and then always runs with the worst-case runtime.

An even more efficient implementation has been proposed by Liu et al. [LSR+15] who introduce an assembly optimized shifting-addition-multiplication-subtraction-subtraction algorithm (SMAS2). They compute a modular multiplication in 53 clock cycles for $q = 7681$, which beats the subtract-and-shift approach mentioned above. For all reductions modulo $q = 7681$ and $q = 12289$ we thus rely on the SMAS2 method.

Extraction of Stages

As additional optimization we use specific routines for the first and the last stage of each NTT. A common optimization is to recognize that $\omega^0 = 1$ in the first stage of the $\mathrm{NTT}^{CT}_{no\rightarrow bo}$ so that only additions are required. As we merge the multiplication by powers of ψ into the NTT this is not the case anymore (see Figure 6.1). However, it is still beneficial to write a specific loop that performs the first stage of the $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ and the last stage of the $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ transformation to achieve less loop overhead (simpler address generation) and less loads and stores.

Usage of Look-up Tables for Narrow Input Distributions

As discussed in Section 6.2.3 it is common in lattice-based cryptography to apply forward transformations mostly to values sampled from a narrow Gaussian error/noise distribution (other polynomials are usually constants and pre-computed). In this case only a limited number of inputs to the butterfly of the first stage of the $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ transformation exist and a pre-computed look-up table can be used instead of modular multiplications. The range of possible input coefficients to the first stage butterfly is rather small, since they are Gaussian distributed and bounded by $[-\tau\sigma, \tau\sigma]$ for standard deviation σ and tail-cut factor τ. Additionally, we only store the result of multiplications of two positive factors. That means that for negative inputs we invert before the look-up and again after the look-up. The same approach would also work for the binary error distribution used for the IBE scheme in [DLP14] and it would be possible to cover even two or more stages due to the very limited input range.

6.3.2 Gaussian Sampling Based on the CDT-Approach

Our implementation of $\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{enc}}$ and $\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{dec}}$ of the RLWEenc scheme, as described in Section 3.4, mainly consists of forward and inverse NTT transformations (NTT and INTT), Gaussian sampling (SampleGaussPoly), and point-wise multiplication (Pointwise, also denoted as $\circ$). We assume that secret and public keys are stored in the read-only flash memory, but loading from RAM would also be possible, and probably even faster. Due to the usage of the NTT, the $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ transformation is only applied on the Gaussian distributed polynomials $\mathbf{e}_1, \mathbf{e}_2$. Thus we can optimize the transformation for this input distribution and with either $\sigma = 4.52$ or $\sigma = 4.85$ it is possible to substitute approx. 99.96% of the multiplications in stage one of $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ by look-ups to a table of 16 entries that requires only 32 bytes of flash memory. For the sampling of the Gaussian distributed polynomials with high precision[2] we use a cumulative distribution table (CDT) [Dev86, DG14]. We construct the table M with entries $p_z = \Pr(x \leq z : x \leftarrow D_\sigma)$ for $z \in [0, \tau\sigma]$ with a precision of $\lambda = 128$ bits. The tail-cut factor τ determines the number of entries $|z_t| = \lceil \tau\sigma \rceil$ of the table and reduces the output of the final sampler to the range $x \in \{-\lceil \tau\sigma \rceil, \ldots, \lceil \tau\sigma \rceil\}$.

To sample a value we choose a uniformly random y from the interval $[0, 1)$ and a bit b and return the integer $(-1)^b z \in \mathbb{Z}$ such that $y \in [p_{z-1}, p_z)$. As we store only the positive half of the table and then sample a sign bit the probability of sampling zero has been pre-halved when constructing the table. For efficiency reasons we just work with the binary expansion of the fractional part instead of floating point arithmetic as all numbers used are smaller than 1.0. The constant CDT matrix M is stored in the read-only flash memory with $k = \lceil \sigma\tau \rceil$ rows and $l = \lceil \lambda/8 \rceil$ columns. In order to sample a Gaussian distributed value we perform a linear search in the table to obtain z. Another option would be binary search, however, for this table size with $x \leftarrow D_\sigma$ being small, the evaluation can already be stopped after only a few comparisons with

---

[2] It is debatable which precision is really necessary in RLWEenc and what impact less precision would have on the security of the scheme, e.g., $\lambda = 40$. But as the implementation of the CDT for small standard deviations σ is rather efficient and for better comparison with related work like [BJ14, BSJ14, dCRVV15] we chose to implement high precision sampling and set $\lambda = 128$.

high probability. The test if the random y is in the range $[p_{z-1}, p_z)$ is performed in a lazy manner on bytes. In this terms laziness means that a comparison is finished when the first bit (or byte on our 8-bit architecture) has been found that differs between two values. Thus we do not need to sample the full λ bits of y and obtain the result of the comparisons early. Random numbers are obtained from a pseudo random number generator (PRNG) using the hardware AES-128 engine running in counter mode. The PRNG is seeded by noise from the least significant bit of the analog digital converter. For the state (key, plaintext) 32 bytes of statically allocated memory are necessary. The final table size is 624 bytes for $\sigma = 4.52$ ($q = 7681$) and 672 bytes for $\sigma = 4.85$ ($q = 12289$).

6.3.3 Results

All implementations are measured on an ATxmega128A1 8-bit microcontroller running at 32 MHz and featuring 128 Kbytes read-only flash, 8 Kbytes RAM and 2 Kbytes EEPROM. Cycle accurate performance measurements were obtained using two coupled 16-bit timer/counters and dynamic RAM consumption is measured using stack canaries. All public and private keys are assumed to be stored in the flash of the microcontroller and we consider the `.text + .data + .bootloader` sections to determine the flash memory utilization. For our implementation we used no calls to the standard library, the avr-gcc compiler in version 4.7.0, and the following compiler options (shortened): `-Os -fpack-struct -ffunction-sections -fdata-sections -flto`.

Detailed cycle counts for the encryption and decryption as well as the most expensive operations are given in Table 6.1. The runtime of encryption is dominated by the NTT (two calls of $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ and one call of $\mathrm{INTT}^{GS}_{bo\rightarrow no}$) which requires approx. 62% of the overall cycles for the RLWEenc-Ia parameter set. The Gaussian sampling requires 29% of the overall cycles which is approx. 328 (RLWEenc-Ia) or 334 (RLWEenc-IIa) cycles per sample. The reason that $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ is slightly faster than $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ is that we use table look-ups for the first stage of the forward transformation (see Section 6.3.2). The remaining amount of cycles (9%) is consumed by additions, point-wise multiplications by a constant/key stored in the flash (PwMulFlash), and message encoding (Encode). In Table 6.1 we also list cycle counts of operations that are now obsolete, especially BitRev and PowMul. For PowMul we assume an implementation where the powers of ψ are computed on-the-fly to save flash memory, otherwise the costs are the same as PwMulFlash. Decryption is extremely simple, fast, and basically calls $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$, the decoding and an addition so that roughly 148 decryption operations could be performed per second on the ATxmega128. Note that we also evaluated RLWEenc-Ib, RLWEenc-IIb, and RLWEenc-IIIb using classic schoolbook multiplication and Karatsuba multiplication and provide results in Section 6.4 and Section 6.5, respectively. However, it turned out that these algorithms cannot beat the NTT and might only be advantageous when extremely small code size is required.

6.4 Implementation of RLWEenc Using the Schoolbook Algorithm

The advantage of the schoolbook polynomial multiplication algorithm is its simplicity, possibility of in-place modular reduction modulo $\mathbf{x}^n + 1$, and that it works for any parameters n and q of the multiplied polynomials. Different variants of the schoolbook algorithm are covered in Section 2.4.1 but the obviously the drawback of this method is the quadratic time complexity.

Table 6.1: Cycle counts and flash memory consumption in bytes of our implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the NTT.

Operation	(n=256, q=7681)	(n=512, q=12289)
	Cycle counts and stack usage	
$\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{enc}}$	874,347 (109 bytes)	2,196,945 (102 bytes)
$\mathsf{RLWEenc}^{\mathsf{NTT}}_{\mathsf{dec}}$	215,863 (73 bytes)	600,351 (68 bytes)
$\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$	185,360	502,896
$\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$	168,853	427,827
SampleGaussPoly	84,001	170,861
PwMulFlash	22,012	53,891
AddEncode	16,884	37,475
Decode	4,407	8,759
	Cycle counts of obsolete functions	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$	198,491	521,872
BitRev	29,696	75,776
BitrevDual	32,768	79,872
PowMul_{ψ}	35,068	96,603
	Static memory consumption in bytes	
Complete binary	6,668	9,258
RAM	1,088	2,144

The stack usage is divided into a fixed amount of memory necessary for plaintext, ciphertext, and additional components (like random number generation) and the dynamic consumption of the encryption and decryption routine. We encrypt a message of n bits.

However, for some scenarios it was demonstrated in works like [GPW$^+$04,HW11] that variants of this method can still be very competitive compared to theoretically more efficient multiplication techniques. In this section we thus provide an implementation of RLWEenc public-key encryption using the schoolbook multiplication to investigate whether this is also the case for lattice-based cryptography. We use the same components for message encoding and Gaussian sampling as in Section 6.3.

6.4.1 Implementation

We realized the modular multiplication and reduction modulo $q = 4093$ (`mm4093`) in assembly exploiting that $2^{12} \mod 4093 = 3$. The multiplication by 3 is realized by shift and addition operations. As we have chosen to represent $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ as a polynomial with unsigned coefficients in $[0, q)$ we do not have to deal with more complicated signed arithmetic and reduction. By carefully assigning the available 8-bit registers according to the compiler's calling convention we ensure that no `push` and `pop` instructions are necessary in `mm4093`. The inner loop of the polynomial multiplication routine implemented in assembly mainly consist of `ld` and `lpm` instructions to read the first coefficient from RAM and the second coefficient from the flash. A subsequent call to `mm4093` performs the modular multiplication and a branch determines whether $i + j > n$ so that the result has to be subtracted or just added to the memory address $i + j \mod n$. The

Table 6.3: Cycle counts and flash memory consumption (in bytes) of our implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the schoolbook algorithm.

Operation	(n=192, q=4093)	(n=256, q=4093)	(n=320, q=4093)
	Cycle counts and stack usage		
encrypt	7,294,976 (446 bytes)	12,289,025 (573 bytes)	21,684,224 (705 bytes)
decrypt	3,205,121 (444 bytes)	5,491,712 (571 bytes)	9,822,330 (703 bytes)
SchoolMul	3,196,529	5,480,074	9,807,218
SampleGaussPoly	55,296	73,727	89,088
AddEncode	12,773	16,915	20,707
Decode	3,331	4,419	5,507
	Static memory consumption in bytes		
Flash	4,648	5,032	5,418
RAM	792+32	1,056+32	1,320+32

implementation of the encryption procedure consists of three calls to the Gaussian sampler, two calls to SchoolMul, and one call to the AddEncode routing that merges the threshold encoding step with an addition.

6.4.2 Results

Results for schoolbook multiplication given in Table 6.3 show that for $n = 192$ an encryption operation takes more than seven million cycles and that approx. 90% of the computation time is spent on the two polynomial multiplications (SchoolMul). All other operations like addition and decoding have only a marginal impact on the runtime. Larger parameter sets turned out to be impractical, e.g., the largest variant for $n = 320$ already consumes more than 21.7/9.8 million cycles for encryption/decryption. The RAM consumption is mainly dominated by the need to be able to store one temporary polynomial which accounts for $2n$ bytes of memory on the stack. However, improvements might be possible when q is chosen as a power of two (see [BLP+13, PG14]) due to even simpler reduction. In our implementation the reduction modulo 4093 accounts for approx. 25 cycles in `mm4093` (called $\approx n^2$ times) but schoolbook multiplication still does not seem to be competitive. Additionally, the savings in code size are smaller than expected. For example, the $n = 256$ parameter set implemented using the NTT requires 6,668 bytes of flash memory while our schoolbook implementations needs 5,032 bytes. Thus usage of the schoolbook multiplication algorithm saves only 1,636 bytes of flash memory.

6.5 Implementation of RLWEenc Using Karatsuba's Algorithm

Polynomial multiplication with improved complexity of $\mathcal{O}(n^{\log(3)})$ operations in $\mathbb{Z}_q$ can be achieved with Karatsuba's method [KO63]. The idea behind this method is to trade one expensive multiplication for three cheap additions by means of the divide and conquer principle. Our implementation of polynomial multiplication applies the Karatsuba method up to a configurable number of iterations. In practice it is reasonable to limit the recursion to a certain degree in order to avoid too much computations due to the increasing number of additions. The remaining

Table 6.4: Cycle counts and flash memory consumption (in bytes) of our implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the Karatsuba algorithm.

Operation	(n=192, q=4093)	(n=256, q=4093)	(n=320, q=4093)
	Cycle counts and stack usage		
encrypt	2,858,803 (2,143 bytes)	4,467,168 (2,760 bytes)	6,426,197 (3,392 bytes)
decrypt	1,336,957 (2,139 bytes)	2,108,398 (2,756 bytes)	3,054,861 (3,388 bytes)
	Static memory consumption in bytes		
Flash	5,920	6,304	6,690
RAM	792+34	1,056+34	1,320+34

low-degree polynomial multiplications are then performed with the schoolbook algorithm. In case the degree of the input polynomials is not a power of two, we apply zero padding.

6.5.1 Implementation

Implementations of Karatsuba's algorithm on constrained devices show typically the drawback of a significant memory consumption, especially, with extensive recursions. We initially aim at reducing memory allocation and thus reserve $2n$ words to store the result of a multiplication and $\sum_{i=0}^{r} \frac{n}{2^i}$ words of additional SRAM storage, depending on the number of recursions r. Immediate reallocation of memory is performed after a recursion step has been completed. Since the degree of the polynomial is then twice as large as the degree of the intermediate polynomials in this recursion we can use the previously disposed memory as a temporary storage for further intermediate polynomials.

For the modular multiplications of individual coefficients we apply the assembler implementation `mm4093` introduced in Section 6.4. We also measure the impact of changing the parameter q from 4093 to 4096[3] and implement a `mm4096` function to exploit the fact that 4096 is a power of two.

6.5.2 Results

In Table 6.4 and Table 6.5 we provide cycle counts of an implementation of RLWEenc where polynomial multiplication is realized with the Karatsuba algorithm. We experimentally evaluated that six levels of recursion lead to the best runtime for all dimensions. With roughly 2.9 million cycles for one encryption and 1.3 million cycles for decryption, this implementation outperforms schoolbook multiplication and is also able to deal with larger values of n. With $q = 4096$ the performance is even better and also the flash memory consumption decreases since we do not need a complex modular reduction function anymore. However, the recursive nature of the Karatsuba algorithm and the need to reduce the polynomial modulo $\mathbf{x}^n + 1$ after it has been completely multiplied leads to significant dynamic RAM consumption (i.e., stack usage).

---

[3] Note that the impact on security is currently not clear, but we still see it as an interesting experiment. We also refer to [PG14] for a description of a hardware implementation of RLWEenc with q being a power of two and a brief discussion on this choice.

Table 6.5: Cycle counts and flash memory consumption (in bytes) of our implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the Karatsuba algorithm with a modified parameter q=4096.

Operation	(n=192, q=4096)	(n=256, q=4096)	(n=320, q=4096)
	Cycle counts and stack usage		
encrypt	2,158,343 (2,128 bytes)	3,240,959 (2,820 bytes)	4,528,167 (3,458 bytes)
decrypt	994,944 (2,180 bytes)	1,508,089 (2,818 bytes)	2,124,268 (3,456 bytes)
	Static memory consumption in bytes		
Flash	4,956	5,340	5,726
RAM	792+34	1,056+34	1,320+34

6.5.3 Comparison with Related Work

A detailed comparison of our implementation with related work that also targets the AVR[4] platform is given in Table 6.6. Our implementation of RLWEenc-Ia encryption outperforms the software from [BSJ14] by a factor of 2.3 and results from [BJ14] by a factor of 3.8 in terms of cycle counts. Decryption is 3.6 times and 6.5 times faster, respectively.

Compared with the 80-bit secure McEliece cryptosystem based on QC-MDPC codes [HvMG13] we get 31 times less cycles for the encryption and even 402 times less cycles for decryption. Translating the implementation results for RSA and ECC given in [GPW$^+$04] to cycle counts, it turns out that an ECC secp160r1 operation requires 6.5 million cycles. RSA-1024 encryption with public key $e = 2^{16} + 1$ takes 3.44 million cycles [GPW$^+$04] and RSA-1024 decryption using the Chinese remainder theorem (CRT) requires 75.68 million cycles (this number is taken from [LGK10]). A comparison with NTRU implementations is currently not easily possible due to lack of published results for the AVR platform[5].

We also refer to [dCRVV15] for an implementation of RLWEenc-Ia and RLWEenc-IIa on an ARM Cortex-M4 (32-bit, 168 MHz). The authors especially make use of the 32-bit wide registers (e.g., to load two coefficients with memory access) but a comparison across architectures with different bit-widths is naturally hard.

6.6 Conclusion and Future Work

Our results indicate that the NTT is an efficient and suitable algorithm for high-performance implementation of the polynomial multiplications required by RLWEenc on an 8-bit AVR platform. The implementations of RLWEenc using Schoolbook and Karatsuba's algorithms are significantly slower compared to the NTT and the savings in code size are smaller than expected.

In future work, a certain speedup seems possible by implementing more parts of RLWEenc and polynomial multiplication in assembly. Moreover, the most crucial operation is the reduction modulo q and we expect the possibility of additional savings through different techniques. Some

---

[4] While the ATxmega128 and ATxmega64 compared to the ATmega64 differ in their operation frequency and some architectural differences cycle counts are mostly comparable.

[5] One exception is a Master thesis by Monteverde [Mon08], but the implemented NTRU251:3 variant is not secure anymore according to recent recommendations in [HHHW09].

Table 6.6: Comparison of our AVR implementation of the NTT and RLWEenc with related work.

Scheme	Device	Operation	Cycles		OP/s	
RLWEenc-Ia ($n = 256$)	AX128	Enc/Dec	874,347	215,863	36.60	148.24
RLWEenc-IIa ($n = 512$)	AX128	Enc/Dec	2,196,945	600,351	14.57	53.30
RLWEenc-Ia ($n = 256$) [LSR+15]	AX128	Enc/Dec	671,628	275,646	48.65	116.09
RLWEenc-IIa ($n = 512$) [LSR+15]	AX128	Enc/Dec	2,617,459	686,367	12.23	46.62
RLWEenc-Ia ($n = 256$) [BSJ14]	AT64	Enc/Dec	3,042,675	1,368,969	2.63	5.84
RLWEenc-Ia ($n = 256$) [BJ14]	AX64	Enc/Dec	5,024,000	2,464,000	6.37	12.99
$\text{NTT}^{CT,\psi}_{no\rightarrow bo}$ ($n = 256$)	AX128	NTT	198,491		161.21	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [LSR+15]	AX128	NTT	193,731		165.17	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [BSJ14]	AT64	NTT	754,668		10.60	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [BJ14]	AX64	NTT	1,216,000		26.32	
$\text{NTT}^{CT,\psi}_{no\rightarrow bo}$ ($n = 512$)	AX128	NTT	521,872		61.31	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [LSR+15]	AX128	NTT	441,572		72.47	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [BSJ14]	AT64	NTT	2,207,787		3.62	
$\text{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [BJ14]	AX64	NTT	2,752,000		11.63	
QC-MDPC McEliece [HvMG13]	AX128	Enc/Dec	26,767,463	86,874,388	1.20	0.37
RSA-1024 [GPW+04]	AT128	Enc/Dec	3,440,000	87,920,000	2.33	0.09
RSA-1024† [LGK10]	AT128	priv. key	75,680,000		0.11	
Curve25519† [DHH+15]	AT2560	Point mul.	13,900,397		1.15	
ECC-ecp160r1 [GPW+04]	AT128	Point mul.	6,480,000		1.23	

By AX128 we identify the ATxmega128A1 clocked with 32 MHz, by AX64 the ATxmega64A3 clocked with 32 MHz, by AT64 the ATmega64 clocked with 8 MHz, by AT128 the ATmega128 clocked with 8 MHz, and by AT2560 the ATmega2560 clocked with 16 MHz. Implementations marked with (†) are claimed to be resistant against timing attacks.

of the proposed future work regarding authenticated key exchange (AKE) and unauthenticated key exchange discussed in Section 5.6 could also be done on the AVR or a different constrained device. Additionally, different or more specialized polynomial multiplication algorithms could lead to further performance improvements (e.g., Nussbaumer multiplication [Nus80, Nus82]). Another natural future work is protection against side-channel attacks, especially timing side-channels, and implementation of conversions that achieve security against chosen ciphertext attacks.

Chapter 7

Lattice-Based Signatures on Reconfigurable Hardware

In this chapter we investigate the efficiency of two post-quantum signature schemes on reconfigurable hardware. As the first contribution we describe a flexible core that can be used to perform the signing and verification operations of the GLP *[GLP12] signature scheme. Additionally, we look at the efficiency of the bimodal lattice signature scheme (*BLISS*) [DDLL13a] and present an implementation of an efficient discrete Gaussian sampler based on a cumulative distribution table (CDT) and convolutions of Gaussians. Both schemes rely on the previously discussed fast NTT microcode engine, use a parallel multiplier for sparse polynomial multiplication, and instantiate the Quark (*GLP*) or Keccak (*BLISS*) hash functions. The implementation of* GLP *presented in this chapter was published in [GLP15] and is an extension and redesign of the first implementation and proposal of* GLP *in [GLP12]. This major redesign and extension justifies an inclusion into this thesis as the implementation in [GLP12] was mainly derived from the author's diploma thesis [Pöp11]. The description of our* BLISS *implementation is based on work published in [PDG14a] and its extended version [PDG14b].*

Contents of this Chapter

7.1 Introduction

For a long time, lattice constructions and especially lattice-based signatures have only been considered secure for inefficiently large parameters that are well beyond practicability (e.g., [Lyu09]) or were, like GGH [GGH97] and NTRUSign [HHP$^+$03], broken due to flaws in the ad-hoc design approaches [NR09, DN12]. This has changed since the introduction of cyclic and ideal lattices [Mic07] and related computationally hard problems like RSIS [PR06, LM06, LMPR08] and RLWE [LPR10b] (see Section 2.3). Especially, constructions that use the Fiat-Shamir paradigm [FS86] have led to a family of fast signature schemes with reasonable signature and key sizes [Lyu09, Lyu12, GLP12, DDLL13a, BG14].

First efforts to tune an ideal lattice-based signature scheme for efficiency on constrained devices or hardware were made in 2012 by Güneysu et al. [GLP12] who proposed the GLP signature scheme (see Section 3.5). The small signatures and good performance were achieved by a new signature compression mechanism, a more "aggressive", non-standard hardness assumption, and the decision to use uniform (as in [Lyu09]) instead of Gaussian noise to hide the secret key contained in each signature via rejection sampling. Additionally, the scheme features a simple key generation mechanism that only requires uniform sampling of small polynomials and polynomial multiplication.

While GLP allows high performance on low-cost FPGAs [GLP12] it later turned out that the scheme is suboptimal in terms of signature size and its claimed security level compared to the bimodal lattice signature scheme (BLISS) [DDLL13a]. The main reason for this is that Gaussian noise, which is prevalent in almost all lattice-based constructions, allows more efficient, more secure, and also even smaller signatures. However, while other techniques relevant for lattice-based cryptography, like fast polynomial arithmetic on ideal lattices, received some attention (see Chapter 4), it is currently not clear how efficient Gaussian sampling can be done on reconfigurable and embedded hardware for large standard deviations. Results targeting signal processing applications (e.g., [TLLV07, GTV12]) are not directly applicable, as usually continuous Gaussians are considered and as the adaption of these algorithms for the discrete case is not trivial (see, e.g., [BCG+13] for a discrete version of the Ziggurat algorithm). First progress was recently made by Roy et al. [RVV13] based on work by Galbraith and Dwarakanath [DG14] but the implemented sampler is only evaluated for very low standard deviations commonly used for lattice-based encryption. Straightforward implementations of fast table-based discrete Gaussian sampling lead to rather large tables, e.g., 40 to 50 KB on an ATxmega64A3 [BJ14], so that more work is required to make lattice-based signature schemes efficient on constrained devices. Despite the issue with inefficiencies caused by Gaussian sampling, BLISS seems to be currently the most promising scheme with a signature length of 5,600 bits, equally large public keys, and 128-bit of equivalent symmetric security based on a reasonable security assumption. In comparision, signature schemes with explicit reductions to weaker assumptions or standard lattice problems [GPV08, Lyu08a, MP12] currently seem to be too inefficient in terms of practical signature and public key sizes (see [BB13] for an implementation of [MP12]). However, there surely is some room for theoretical improvement, as suggested by the compression ideas developed by Bai and Galbraith [BG14].

7.1.1 Related Work

For an introduction and description of the GLP signature scheme [GLP12] we refer to Section 3.5 while BLISS [DDLL13a] is covered in Section 3.6. Both schemes are based on works by Lyubashevsky [Lyu09, Lyu12]. Other practical signature schemes are BG [BG14], PASS-Sign [HPS+14], a modified NTRU signature scheme [MBDG14], or a signature scheme derived from a recently proposed IBE scheme [DLP14]. Up to our knowledge, no other performance optimized hardware implementations of ideal lattice-based signature schemes have been proposed so far. Implementation results on microcontrollers and CPUs have been published in [GOPS13, OPG14, BSJ14, BJ14, DBG+14, POG15a]. We also refer to [HPO+15] for a survey on lattice-based signatures and their efficient implementation. Techniques for sampling of continuous Gaussians are surveyed in [TLLV07, GTV12] and algorithms for efficient and secure discrete Gaussian sampling are given in [Dev86, GPV08, BCG$^+$13, RVV13, DG14, RRVV14, BCNS15].

7.1.2 Contribution

In this work we examine the efficiency of GLP and BLISS on reconfigurable hardware. Our GLP implementation is fully functional, contains a Trivium-based PRNG [Can06] as well as the lightweight hash function QUARK [AHMN13] and makes use of the extremely efficient NTT (contrary to the schoolbook approach from [GLP12]). For example, on the low-cost Xilinx Spartan-6 we are 1.5 times faster and use only half of the resources of the optimized RSA implementation of Suzuki [Suz07]. With 2,385 signatures and 10,899 signature verifications per second, we can satisfy even high-performance demands with a low area footprint using a Xilinx Virtex-6 device.

To speed up BLISS and other signature schemes that rely on Gaussian sampling and require rather large standard deviations, we implement improved techniques for efficient sampling of discrete Gaussian noise. We realize a sampler based on a CDT where binary search is improved using a shortcut table of intervals and use an optimal data structure that saves roughly half of the table space by exploiting the properties of the Kullback-Leibler divergence (see Section 2.5.4).

Based on these techniques we provide implementations of the BLISS-I, BLISS-III, and BLISS-IV parameter sets on reconfigurable hardware that are tweaked for performance and offer 128 bits to 192 bits of security. For practical evaluation we compare our improvements for the CDT-based Gaussian sampler to the Bernoulli approach presented in [DDLL13a]. Our implementation includes an NTT-based polynomial multiplier, more efficient sparse multiplication, and the KECCAK-f[1600] hash function [BDPA13] to provide the full picture of the performance that can be achieved by employing latest lattice-based signature schemes on reconfigurable hardware. Our BLISS-I implementation on a Xilinx Spartan-6 FPGA supports up to 8,761 signatures per second using 7,193 LUTs, 6,420 flip-flops, 5 DSPs, and 5.5 block RAMs, includes Huffman encoding and decoding of the signature, and outperforms the GLP implementation in [GLP12] in terms of time and area.

7.2 Implementation of GLP

In this section we describe the FPGA implementation of the signing and verification procedures for parameter set I of GLP providing about 80 bits of equivalent symmetric security. We present three implementation variants; BOTH is a combined core for signing and signature verification. The cores named SIGN and VER, however, support signing and verification only, respectively. All three variants are built on top of a single code base and have been extensively tested.[1]

For instantiation of the signature scheme, we used a datapath of 23-bit due to pipelined processing of coefficients of width $\lceil \log_2(q) \rceil$. The discrete Gaussian sampling and polynomial multiplication in step 3 of Algorithm 12 ($\mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$) and step 2 of Algorithm 13 ($\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2' - \mathbf{t}\mathbf{c}$) is performed using the previously discussed microcode engine (see Chapter 4). Four registers

---

[1] Note that we employed a pseudo-random number generator in our implementation for minimal footprint. If required, it can be replaced or seeded by a true random number which was not in the scope of this work (see for example [DG07, Gol06]).

are already fixed where register R0 and R1 are part of the NTT block, R2 is associated to the uniform random sampler while register R3 is exported to upper layers as I/O port. We use R4 to R7 to store temporary values like $\mathbf{y}_1$, $\mathbf{y}_2$, the public constant $\mathbf{a}$ and the public key $\mathbf{t}$. As we use the fine-grained access to NTT instructions, we can exploit that coefficients are fixed most of the time (e.g., constants) or needed twice. We therefore store them directly in NTT representation or transform them only once to save subsequent transformations.

7.2.1 Pipelined Message Signing

The general idea of our implementation is to separate the signing process into three independent blocks which are executed in parallel (see Figure 7.1). These three main blocks are the `Lattice Processor` core, the `Hash` unit implementing the random oracle, and the `Sparse Multiplication` and compression component. Parallel operations are supported by pipelining. As a consequence, input polynomials to the last two blocks are stored in buffers, realized as BRAMs. Thus, we can avoid latencies for inputs $\mathbf{r} = \mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$ to the hash engine and the sparse multiplication. This allows to achieve high throughput and good resource and block utilization. A direct benefit of this approach is that we can choose a moderately fast hash component that just matches the performance of the other two blocks. This enables the use of a resource-efficient implementation of the lightweight hash function Quark [AHMN13].

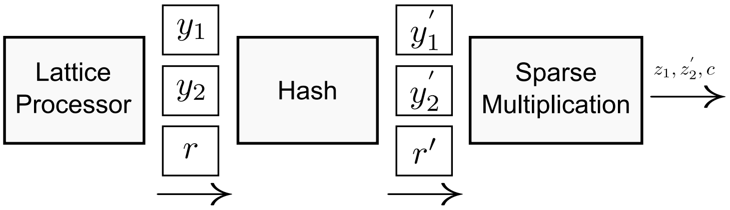

Figure 7.1: Simplified block diagram of our GLP signing engine showing the main blocks `Lattice Processor`, `Hash`, and `Sparse Multiplication`.

The detailed architecture of our signing engine is given in Figure 7.2. We use the lattice processor to sample values $\mathbf{y}_1, \mathbf{y}_2 \xleftarrow{\$} \mathcal{R}_{q,k}$ and directly compute $\mathbf{r} = \mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$ using the NTT. Note that for this operation $\mathbf{a}$ is already stored in NTT representation. As a consequence, the workload consists of sampling $2n$ uniformly random values, one forward and one inverse NTT call as well as some additional overhead for point-wise multiplication and data movement. In signing mode the processor actually starts processing independently of the message or secret key and precomputes triples $(\mathbf{r}, \mathbf{y}_1, \mathbf{y}_2)$ for subsequent use. These triples are stored in a temporary buffer and are accessed during signing by the hashing module. Note also that before $\mathbf{r} = \mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$ can be hashed we have to transform every coefficient into the higher-order representation $\mathbf{r}^{(0)}$. This operation is basically an integer division by the constant 32705 which makes use of the specific bit-layout of the divisor. The hash module is realized by the S-Quark×16 lightweight hash function [AHMN13] implementing the random oracle H. This variant of QUARK offers 224-bit preimage and 112-bit collision security. We employ Quark since it supports high clock frequencies and consumes only few resources. The relatively low throughput just matches the speed of the other pipelining stages of the engine.

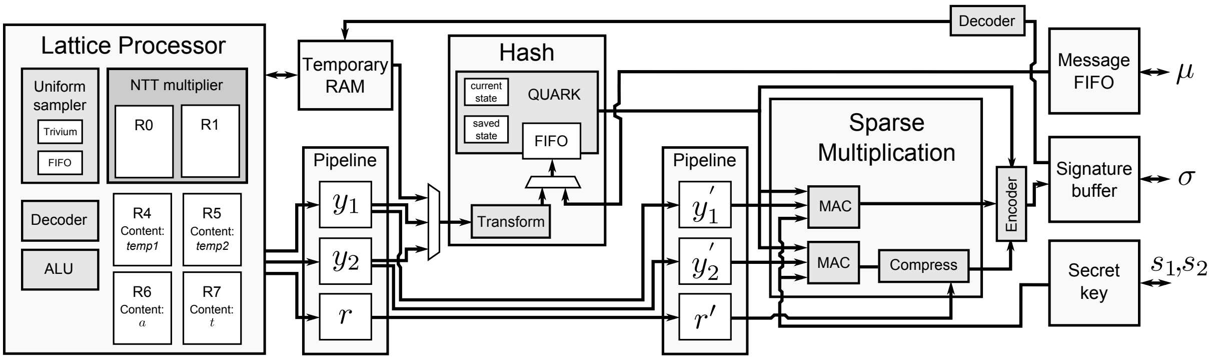

Figure 7.2: Detailed architecture of our GLP signing engine.

In order to allow future extensions and clock domain separation using a FIFO, we have implemented a wrapper around the hash function. Moreover, we have extended the hash in order to be able to save the current state. This is beneficial in the context of rejection sampling. Since on average the hash function has to be restarted seven times, this would imply re-hashing of the message to be signed. In case those messages are long this would require significant additional effort. As a consequence, we first hash the message μ, save the state, hash a binary representation of $\mathbf{r}$ and reload the saved state in case the signature was rejected. This approach is also straightforward in terms of state management, as the message has just to be fed into the hash function once and does not have to be temporarily stored in a RAM or FIFO during the signing process. As S-Quark×16 is a sponge-based construction, we abort the squeezing phase after obtaining 160 output bits to generate $\mathbf{c}$.

The polynomial $\mathbf{c}$, which is computed by the `Hash` unit, and a triple $(\mathbf{r}, \mathbf{y}_1, \mathbf{y}_2)$ are then processed by the sparse multiplication and compression block. Due to the parallelism of the three main blocks, the hash engine can directly request a new triple $(\mathbf{r}, \mathbf{y}_1, \mathbf{y}_2)'$ to generate a new hash/polynomial $\mathbf{c}$. In order to compute $\mathbf{z}_1, \mathbf{z}_2 \in \mathcal{R}_{q,k-32}$ we have to multiply the sparse polynomial $\mathbf{c}$ with the secret key polynomials $\mathbf{s}_1, \mathbf{s}_2 \in \mathcal{R}_{q,1}$ that have coefficients in the range $[-1, 1]$. For this, we implemented a Comba/product-scanning multiplier and compute $\mathbf{z}_1$ and $\mathbf{z}_2$ in parallel. More precisely, we store the secret key polynomials $\mathbf{s}_1, \mathbf{s}_2$ in one block RAM and merged them so that each address holds a coefficient of $\mathbf{s}_1$ and one of $\mathbf{s}_2$ ($\mathbf{s}_1[i]||\mathbf{s}_2[i]$, for $0 \leq i < n$). Since the coefficients of $\mathbf{s}_1, \mathbf{s}_2$ and $\mathbf{c}$ are in $[-1, 0]$, the multiplication can be simply realized by an adder so that the parallel computation of $\mathbf{z}_1, \mathbf{z}_2$ is actually not very resource-intensive. Since the result is returned coefficient by coefficient, this allows immediate rejection when an out-of-range value is detected (i.e., after adding the corresponding coefficient of $\mathbf{y}_1$ or $\mathbf{y}_2$). In this case the signing procedure is directly restarted. In order to prevent memory-intensive expansion of the 160-bit binary hash c into a polynomial, we perform this transformation on-the-fly in the sparse multiplier. The `Compress` component extracts the carry information needed to be able to perform the transformation into the higher-order representation during the signature verification. The component implementing Algorithm 15 maintains a counter to track the number of uncompressed values but does not yield further state information. Returned values are directly encoded and written into an output FIFO. When the signing operation needs to be restarted the FIFO is reset. When successful, however, access to this FIFO is granted to the external interface for retrieving the valid signature.

7.2.2 Signature Verification

The verification algorithm is simpler compared to signing which also results in a smaller resource consumption. Especially, the `Sparse Multiplication` block is obsolete (see Figure 7.1) so that the verification core only consists of the `Lattice Processor` and `Hash` block. The simpler verification directly translates into less computation cycles so that we did not implement pipelining for the two blocks. We first run the `Lattice Processor` to compute $\mathbf{a}\mathbf{z}_1 + \mathbf{z}'_2 - \mathbf{t}\mathbf{c}$ and then activate the hash engine in a next step. Therefore, the runtime for verification consists of the amount of time to perform this computation *and* the time to hash the result.

The validity test if $\mathbf{z}_1, \mathbf{z}'_2 \in \mathcal{R}_{q,k-32}$ is simply being performed during signature decoding from the input FIFO. The main workload is caused by the three polynomial multiplications $\mathbf{a}\mathbf{z}_1, \mathbf{z}'_2, \mathbf{t}\mathbf{c}$ performed by the `Lattice Processor`. With the global constant $\tilde{\mathbf{a}} = \mathsf{NTT}(\mathbf{a})$ and the verification key $\tilde{\mathbf{t}} = \mathsf{NTT}(\mathbf{t})$ stored directly in NTT representation, we finally compute the input to the hash function as follows:

$$\mathsf{INTT}(\tilde{\mathbf{a}} \circ \mathsf{NTT}(\mathbf{z}_1) - \tilde{\mathbf{t}} \circ \mathsf{NTT}(\mathbf{c})) + \mathbf{z}'_2. \tag{7.1}$$

As a consequence, three NTT operations ($\frac{3}{2}n \log n$ cycles), two point-wise multiplications ($2n$ cycles) and two addition/subtractions ($2n$ cycles) have to be performed. The time-consuming decoding of the signature is concurrently performed with the operation of the `Lattice Processor` to avoid further latencies. One design alternative would have been to implement the relatively cheap multiplication $\mathbf{t}\mathbf{c}$ in a separate unit operating in parallel to the lattice processor. This is not too costly as $\mathbf{c}$ is sparse with only 32 coefficients that are either minus one or one. Thus a schoolbook multiplier taking this sparseness into account would only require roughly $32 \cdot 512$ cycles. However, even then the multiplication would be not much faster compared to the NTT, harder to manage by the state machine, and also add resource consumption.

7.2.3 Implementation Aspects

In this section we provide details on the optimization of our implementation to reduce resource consumption or improve flexibility.

Flexible Instantiation

In order to allow flexible usage and maintainability of the design we developed one code base that can be configured by VHDL `generic` statements to generate a core supporting either signing (SIGN), verification (VER) or both operations (BOTH). In this case `if ... generate` statements are used to remove complete unnecessary blocks or just parts of a component (e.g., FSM states). Between the BOTH and the SIGN cores, the small savings in resources stem mainly from removal of the signature decoder and a buffer BRAM. The verification core VER is significantly smaller than BOTH or SIGN since the `Sparse Multiplication` component and the resources for the pipelining stages are not required.

Uniform Sampling

To generate uniformly random distributed coefficients in the range $[-k, k]$ for the polynomials $\mathbf{y}_1, \mathbf{y}_2$ we use rejection sampling in order to obtain a value from the range $[0, \ldots, (2k+1)]$. We therefore compute a coefficient by $c = (r \mod (2k+1)) - k$, resembling the approach that was used for the software implementation in [GOPS13]. The necessary pseudo-random input bits are generated using an implementation of the Trivium stream cipher [Can06][2] which outputs one pseudo-random bit in every clock cycle. In the rejection sampling step, in order to obtain a value from the range $[0, \ldots, (2k+1)]$, we check that a 16-bit random value, interpreted as integer, is not larger or equal to $(2k+1) = 32769$. As a consequence, 50% of the inputs to the sampler are rejected. To still provide sufficient performance, we instantiate three Trivium cores in parallel to extract 3 bits at a time and perform a rejection sampling in $\left\lceil \frac{\lceil \log_2 32769 \rceil}{3} = 6 \right\rceil$ clock cycles on average.

7.2.4 Results

All results below were obtained post place-and-route (PAR) and generated with Xilinx ISE 14.2 and we synthesized the signing and verification engine for the low-cost Spartan-6 (S6SLX25-3) and on the high-speed Virtex-6 (V6LX75T-3) device family. Detailed performance results for parameter set I are given in Table 7.1 and the corresponding resource consumption is addressed in Table 7.3 for Spartan-6 and Table 7.4 for Virtex-6, respectively.

Detailed Performance Evaluation

Table 7.1 provides the actual runtime of several core components of our implementation for a small message. Additionally, we outline theoretical extrapolations based on the parameter n. A difference of roughly five to ten percent between estimation and measurement is due to the need of setup phases, pipeline stalls, or instruction decoding, which are not included in our model.

The runtime of the `Lattice Processor` in signing mode is not constant due to a small amount of wait cycles in the uniform sampler unit. Moreover, the overall runtime of our pipelined implementation is finally limited by the `Lattice Processor` (10,505 cycles) and in case of no early abort, by the `Sparse Multiplication` (worst-case 16,950 cycles). The `Hash` component requires 10,192 cycles to transform and hash $\mathbf{r}$. On average one signing attempt with one input triple takes 15,908 cycles of which about seven attempts are necessary in order to generate a valid signature. The computation of the valid signature requires a total of 114,865 cycles on average. Signature verification involves polynomial arithmetic performed by the processor (13,851 cycles) and hashing of the result (10,192 cycles). The overall verification runs at constant time and takes 25,138 cycles, where the additional 1095 cycles are accounted for initial decoding of the first polynomial and I/O. A significant acceleration of the scheme could be still achieved by using a faster (but larger) hash function as the time to hash contributes roughly 40% to the overall runtime of the verification process.

Resource Consumption and Performance

In Table 7.3 we give post-PAR results for a low-cost Spartan-6 LX25. The figures were obtained after 12 runs of the Xilinx Smart Explorer to achieve the smallest timing score. Based on the maximum clock frequency of 187 MHz, one signing operations takes on average 615 µs while

---

[2] The used implementation of Trivium is based on [Ste]. However, we removed asynchronous reset signals to improve resource utilization.

Table 7.1: Detailed performance evaluation of the main components of our GLP implementation for a short message.

Aspect	Description	Cycles
Lattice processor (sign)	Average amount of cycles to sample $\mathbf{y}_1, \mathbf{y}_2 \overset{\$}{\leftarrow} \mathcal{R}_{q_k}$ and to compute $\mathbf{r} \leftarrow \mathbf{a}\mathbf{y}_1 + \mathbf{y}_2$ (averaged over 400 signatures).	10,505 ($n \log n$+$12n$+23ϵ)
Lattice processor (verify)	Computation of $\mathbf{a}\mathbf{z}_1 + \mathbf{z}_2' - \mathbf{t}\mathbf{c}$	c_{sign}=13,851. ($n \log n + 10n + 18\epsilon$)
Hash	Higher order transformation of $\mathbf{r}$ and evaluation of the QUARK hash function on $\mathbf{r}^{(1)}$ and a short message μ: $\mathsf{H}\left(\mathbf{r}^{(1)}, \mu\right)$.	$c_{hash} = 10{,}192$
Sparse multiplication	Computation of $\mathbf{z}_1 \leftarrow \mathbf{s}_1\mathbf{c} + \mathbf{y}_1, \mathbf{z}_2 \leftarrow \mathbf{s}_2\mathbf{c} + \mathbf{y}_2$, compression, rejection sampling, and signature encoding.	50 – 16,950 (32 to $32n$)
One signing attempt	Average amount of cycles required for one signing attempt (averaged over 400 signatures).	15,908
Signing	Average amount of cycles required to successfully generate a signature (averaged over 400 signatures).	114,865
Verification	Amount of cycles required for the verification of one message.	25,138 ($c_{sign} + c_{hash}$)

Whenever possible we provide actual cycle counts as absolute number and also the theoretical estimation based on the dimension n as well as worst-case and average-case values. Small differences between estimation and measurement are due to wait cycles, pipeline stalls, or setup phases.

verification is constant time and requires 134 µs on the BOTH core. Compared to the combined core for signing and verification, the core supporting signing only saves just a few logic elements while the verification core is much smaller with only 65% LUT and 74% BRAM usage.

In Table 7.4 we provide results for a more expensive but also much faster (in the sense that it supports higher clock frequencies) and larger Virtex-6 LX75. The resource consumption is similar to the Spartan-6 implementation but the achievable clock frequency is nearly 100 MHz higher. As a consequence, we can verify more than 10000 signatures per second with the BOTH and VER core. Note that the Spartan-6 supports 9/18K BRAMs while the Virtex-6 has been designed with larger 18/36K BRAMs. Unfortunately, this leads to some wasted space in BRAMs on the Virtex-6 as our design contains a large number of 9k BRAMs which are mapped to 18k BRAMs on the Virtex-6.

Table 7.3: Performance and resource consumption of all three variants of our GLP implementation targeting a Xilinx Spartan-6 LX25 (speed-grade -3).

Aspect	**BOTH**	**SIGN**-Only	**VER**-Only
Slices	2,045	2,010	1,586
LUT/FF	6,088/6,804	5,614/6,188	3,966/4,318
18K BRAM	19.5	18.5	14.5
DSP48A1	4	4	4
Max clock freq.	187 MHz	197 MHz	187 MHz
Sign (Sig/s)	1,627	1,715	-
Verify (Sig/s)	7,438	-	7,438

Table 7.4: Performance and resource consumption of all three variants of our GLP implementation targeting a Xilinx Virtex-6 LX75T (speed-grade -3).

Aspect	**BOTH**	**SIGN**-Only	**VER**-Only
Slices	2,083	2,086	1,745
LUT/FF	6,591/6,791	6,073/6,183	4,571/4,338
18K BRAM	36	34	26
DSP48E1	4	4	4
Max clock freq.	274 MHz	286 MHz	280 MHz
Sign (Sig/s)	2,385	2,489	-
Verify (Sig/s)	10,899	-	11,138

7.3 Implementation of BLISS

In this section we provide details on our implementation of the BLISS signature scheme on a Xilinx Spartan-6 FPGA. We especially focus on our implementation of discrete Gaussian sampling.

7.3.1 Gaussian Sampling Using Convolutions and a CDT

In this section we present implementation details on the CDT sampler and the Bernoulli sampler proposed in previous work [DDLL13a] to evaluate and validate our results in practice. The theoretical background and analysis of these samplers is discussed in Section 2.5 (see also [PDG14a]).

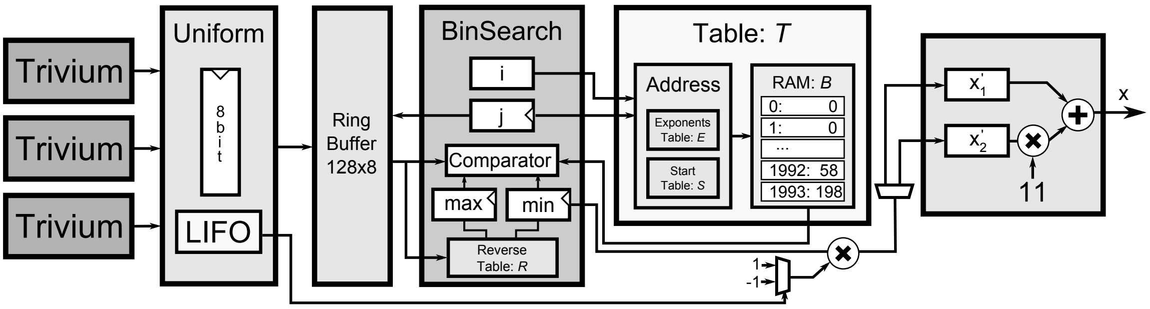

Figure 7.3: Block diagram of our CDT sampler that generates two samples x'_1, x'_2 of standard deviation $\sigma' \approx 19.53$ which are combined to a sample $x = x'_1 + 11x'_2$ with standard deviation $\sigma = 215.73$.

Enhanced CDT Sampling

Our hardware implementation is based on the proposal given in Section 2.5.4 and operates on bytes in order to use the 1024x8-bit mode of operation of the Spartan-6 block RAMs. The design of our CDT sampler is depicted in Figure 7.3 and uses the convolution lemma. Thus two samples with $\sigma' \approx 19.53$ are combined into a sample with standard deviation $\sigma \approx 215.73$. The `BinSearch` component performs a binary search on the table T as described in [PDG14a] for a random byte vector r to find a c such that $T[c] \geq r > T[c+1]$. It accesses T byte-wise and thus $T_j[i] = M_{j-E[i]}[i]$ denotes the entry at index $i \in (0, 261)$ and byte j where $T_j[i] = 0$ when $j - E[i] < 0$ or $i \geq \ell_{j-E[i]}$. When a sampling operation is started in the `BinSearch` component we set $\mathtt{j} = 0$ and initialize the pointer registers `min` and `max` with the values stored in the reverse interval table $R[r_0]$ where r_0 is the first random byte. The reverse interval table is realized as 256x15-bit single port distributed ROM (6 bits for the minimum and 9 bits for the maximum). The index of the middle element of the search radius is $\mathtt{i} = (\mathtt{min}+\mathtt{max})/2$. In case $T_j[\mathtt{i}] > r_{\mathtt{j}}$ we set $(\mathtt{min} = \mathtt{i}, \mathtt{i} = (\mathtt{i} + \mathtt{max})/2, \mathtt{max} = \mathtt{max}, \mathtt{j} = 0)$. Otherwise, for $T_j[\mathtt{i}] < r_{\mathtt{j}}$ we set $(\mathtt{i} = (\mathtt{min} + \mathtt{i})/2, \mathtt{min} = \mathtt{min}, \mathtt{max} = \mathtt{i}, \mathtt{j} = 0)$ until $\mathtt{max} - \mathtt{min} < 2$. In case of $T_{\mathtt{j}}[\mathtt{i}] = r_{\mathtt{j}}$ we increase $\mathtt{j} = \mathtt{j} + 1$ and thus compare the next byte. The actual entries of $M_0 \dots M_8$ are consecutively stored in block memory B and the address is computed as $a = S[j - E[i] + i]$ where we store the start addresses of each byte group in a small additional LUT-based table $S = [0, 262, 524, 759, 982, 1184, 1364, 1521, 1646]$. Some control logic takes care that all invalid/out of bound requests to S and B return a zero.

For random byte generation we use three instantiations of the Trivium stream cipher [Can06] (each `Trivium` instantiation outputs one bit per clock cycle) to generate a uniformly random byte every third clock cycle and store spare bits in a `LIFO` for later use as sign bits. The random values r_j are stored in a 128x8-bit ring buffer realized as simple dual-port distributed RAM. The idea is that the sampler may request a large number of random bytes in the worst-case but usually finishes after one or two comparisons due to the lazy search. As the `BinSearch` component keeps track of the maximum number of accessed random bytes, it allows the `Uniform` sampler to refresh only the used $\max(j) + 1$ bytes in the buffer. In case the buffer is empty, we stop the Gaussian sampler until a sufficient amount of randomness becomes available. In order to

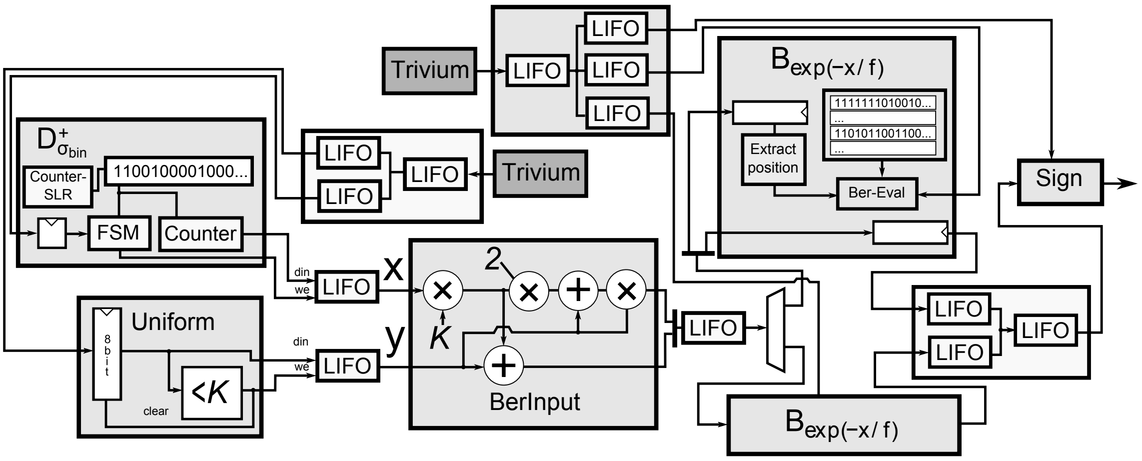

Figure 7.4: Block diagram of the Bernoulli sampler using two instantiations of Trivium as PRNG and two $\mathtt{B}_{\mathtt{exp(-x/f)}}$ components (only one is shown in more detail).

compute the final sample x we assign individual random signs to two samples x_1', x_2' and finally output $x = x_1' + 11x_2'$.

To achieve a high clock frequency, a comparison in the binary search step could not be performed in one cycle due to the excessive number of tables and range checks involved. We therefore allow two cycles per search step which are carefully balanced. For example, we precompute the indices $\mathtt{i}' = (\mathtt{min}{+}\mathtt{i})/2$ and $\mathtt{i}'' = (\mathtt{i}{+}\mathtt{max})/2$ in the cycle prior to a comparison to relax the critical paths. Moreover, timing was improved by not performing checks on the validity of the address used to access the block RAM but by just zeroing out the output in case the address was invalid. Note also that we are still accessing the two ports of the block RAM holding B only every two clock cycles which would enable another sampler to operate on the same table using time-multiplexing. Implementations of the CDT sampler for $\sigma \approx 250.54$ and $\sigma \approx 271.93$ just differ by different tables and constants which are selected during synthesis using a `generic` statement.

Bernoulli Approach

In [DDLL13a] Ducas et al. proposed an efficient Gaussian sampling algorithm which can be used to lower the size of precomputed tables to $\lambda \log_2(2.4\tau\sigma^2)$ bits without the need for long-integer arithmetic and with low entropy consumption ($\approx 6 + 3\log_2 \sigma$). A description and additional background on the sampler is contained in Section 2.5.3. The general advantage of this sampler is a new technique to reduce the probability of rejections by first sampling from an (easy to sample) intermediate distribution and then from the target distribution.

The block diagram of the implemented sampler is given in Figure 7.4. In the $\mathtt{D}^+_{\sigma_{\mathrm{bin}}}$ component a $x \in D^+_{\sigma_{\mathrm{bin}}}$ is sampled according to Algorithm 4. However, on-the-fly construction of the binary distribution of $\rho_{\sigma_{\mathrm{bin}}}(\{0, \dots, j\}) = 1.1001000010000001...$ (see Section 2.5.3) is not necessary as we use two 64-bit shift registers (LUTM) to store the expansion precomputed up to a precision of 128 bits. Uniformly random values $y \in \{0, \dots, k-1\}$ are sampled in the `Uniform` component using rejection sampling (for $k = 254$ with $\frac{2}{256}$ the probability of a rejection is low[3]). The pipelined `BerInput` component takes a (y, x) tuple as input and computes $t = kx$ and outputs $z = t + y$ as well as $j = y(y + 2t)$. While z is retained in a register, the $\texttt{B}_{\texttt{exp(-x/f)}}$ module evaluates the Bernoulli distribution of $b \leftarrow B_{\exp(-j/2\sigma^2)}$. Only if $b = 1$ the value z is passed to the output and discarded otherwise. The evaluation of $B_{\exp(-x/f)}$ requires independent evaluations of Bernoulli variables. Sampling from $\mathcal{B}_c$ is easy and can be done by just evaluating $s < c$ for a uniformly random $s \in [0, 1)$ and a precomputed c. The precomputed tables $c_i = \exp(-2^i/f)$ for $0 \leq i \leq l, f = 2\sigma^2$ where l is $\lceil \log_2(\max(j)) \rceil$ are stored in a distributed RAM. The $\texttt{B}_{\texttt{exp(-x/f)}}$ module (Algorithm 3) then searches for one-bit positions u in j and evaluates the Bernoulli variable B_{c_u}. This is done in a lazy manner so that the evaluation aborts when the first bit has been found that differs between a random s and c. This techniques saves randomness and also runtime. As the chance of rejection is larger for the most significant bits we scan them first in order to abort as quickly as possible. In the last step the `Sign` component samples a sign bit and rejects half of the samples where $z = 0$.

The Bernoulli sampler is suitable for hardware implementation as most operations work on single bits (mostly comparisons) only. However, due to the non-constant time behavior of rejection sampling we had to introduce buffers between each element (see Figure 7.4) to allow parallel execution and maximum utilization of every component. This includes the distribution and buffering of random bits. In order to reduce the impact of buffering on resource consumption we included Last-In-First-Out (LIFO) buffers that solely require a single port RAM and a counter as the ordering of independent random elements does not need to be preserved by the buffer (what would be the case with a FIFO). For maximum utilization we have evaluated combinations of sub-modules and finally implemented two $\texttt{B}_{\texttt{exp(-x/f)}}$ modules fed by two instantiations of the Trivium stream cipher to generate pseudo random bits. A detailed analysis is given in Section 7.3.4.

7.3.2 Design of a Signing and of a Verification Core

The architecture of our implementation of a high-speed BLISS signing engine is given in Figure 7.5 and the same for all supported parameter sets (I,III,IV). Similar to the GLP design in [GLP12] we implemented a two stage pipeline for signing where the polynomial multiplication $\mathbf{a}_1\mathbf{y}_1$ runs in parallel to the hashing $H(\lfloor \mathbf{u} \rceil_d, \mu)$ and sparse multiplication $\mathbf{z}_1 = \mathbf{s}_1\mathbf{c} \pm \mathbf{y}_1$ and $\mathbf{z}_2 = \mathbf{s}_2\mathbf{c} \pm \mathbf{y}_2$. Another option would be a three stage pipeline, as in the GLP implementation given in Section 7.2, with an additional buffer between the hashing and sparse multiplication. As a trade-off this would allows to use a slower and thus more area efficient hash function but also imply a longer delay and require pipeline flushes in case of an accepted signature.

Polynomial Multiplication

For FFT/NTT-based polynomial multiplication of $\mathbf{a}_1\mathbf{y}_1$ we rely on the microcode engine discussed in Chapter 4. As the public key $\mathbf{a}_1$ is already stored in NTT format, for BLISS we just have to perform a sampling operation, a forward transformation, point-wise multiplication, and one inverse transformation. In contrast to the conference version [PDG14a] we thus removed

---

[3] Rejection sampling could be avoided completely by setting $k = 256$ and thus by sampling using $\sigma = k\sigma_{\text{bin}} \approx 217.43$. However, we decided to stick to the original parameter as the costs of rejection sampling are low.

unnecessary flexibility of the core (e.g., polynomial addition or subtraction), fixed some generic options to $(n = 512, q = 12289)$, and also support only one NTT enabled register (instead of two). Special NTT registers are necessary in the multiplier implementation to provide two write and two read ports required by the NTT butterfly (see [RVM+14] for an improvement of this aspect). As the registers are comprised of two block RAMs, which are only filled with $n/2$ coefficients (thus $n/2 \cdot \log_2(q) = 3584$ bits), this saves one block memory.

Hash Block

When a new triple $(\mathbf{a}_1\mathbf{y}_1, \mathbf{y}_1, \mathbf{y}_2)$ is available the data is transferred into the block memories `BRAM-U`, `BRAM-Y1` and `BRAM-Y2` and the small polynomial $\mathbf{u} = \zeta\mathbf{a}_1\mathbf{y}_1 + \mathbf{y}_2$ is computed on-the-fly and stored in `BRAM-U` for later use. The lower order bits $\lfloor \mathbf{u} \rceil_d \bmod p$ of $\mathbf{u}$ are saved in the `RAM-U`. As random oracle instantiation we have chosen the KECCAK-f[1600] hash function for its security and speed in hardware [SSRG11, JA11]. A configurable hardware implementation[4] is provided by the KECCAK project and the `mid-range` core is parametrized so that the KECCAK state is split into 16 pieces ($Nb = 16$). To simplify control logic and padding we just hash multiples of 1024-bit blocks and rehash in case of a rejection. Storing the state of the hash function after hashing the message (and before hashing $\lfloor \mathbf{u} \rceil_d \bmod p$) would be possible but is not done due to the state size of KECCAK. After the hash generation, the `ExtractPos` component extracts the κ positions of $\mathbf{c}$ which are one from the binary hash output and stores them in the 23x9-bit memory `RAM-Pos`.

Sparse Multiplication

For the computation of $\mathbf{z}_1' = \mathbf{s}_1\mathbf{c}$ and $\mathbf{z}_2' = \mathbf{s}_2\mathbf{c}$ we then exploited that $\mathbf{c}$ has mainly zero coefficients and only $\kappa = 23$ coefficients set to one. Moreover, only $d_1 = \lceil \delta_1 n \rceil = 154$ coefficients in $\mathbf{s}_1$ are ± 1 and $\mathbf{s}_2$ has d_1 entries in ± 2 where the first coefficient is from $\{-1, 1, 3\}$. The simplest and, in this case, also best suited algorithm for sparse polynomial multiplication is the row- or column-wise schoolbook algorithm (see Section 2.4.1). While row-wise multiplication would benefit from the sparsity of $\mathbf{s}_{1,2}$ *and* $\mathbf{c}$, more memory accesses are necessary to add and store inner products. Since memory that has more than two ports is extremely expensive, this also prevents efficient and configurable parallelization. As a consequence, our implementation consists of a configurable number of cores (C) which perform column-wise multiplication to compute $\mathbf{z}_1$ and $\mathbf{z}_2$, respectively. Each core stores the secret key (either $\mathbf{s}_1$ or $\mathbf{s}_2$) efficiently in a distributed RAM and accumulates inner products in a small multiply-accumulate unit (`MAC`). Positions of $\mathbf{c}$ are fed simultaneously into the cores. Another advantage of our approach is that we can compute the norms and scalar products for rejection sampling parallel to the sparse multiplication. In Figure 7.5 a configuration with $C = 2$ is shown for simplicity but our experiments show that $C = 8$ leads to a good trade-off between speed and resource consumption.

Signature Verification

Our verification engine uses only the `PolyMul` (without a Gaussian sampler) and the `Hash` component and is thus much more lightweight compared to signing. The polynomial $\mathbf{c}$ stored as

---

[4] See `http://keccak.noekeon.org/mid_range_hw.html` for more information on the core.

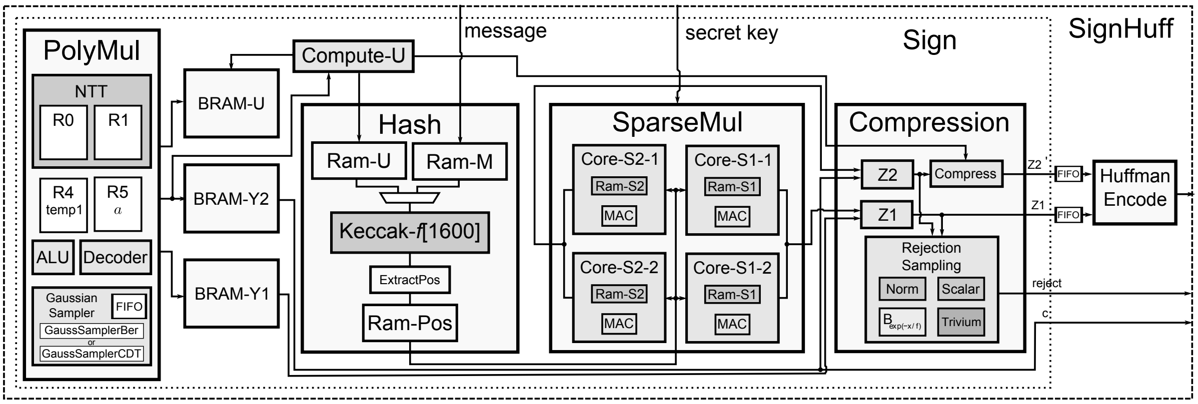

Figure 7.5: Block diagram our BLISS-I signing engine.

(unordered) positions and is expanded into a 512x1-bit distributed RAM and the input to the hash function is computed in a pipelined manner when `PolyMul` outputs $\mathbf{a}_1\mathbf{y}_1$.

7.3.3 Huffman Encoding for Short Signatures

The `Sign` and `Verify` components described above operate on signatures $(c, \mathbf{z}_1, \mathbf{z}_2^\dagger)$ that consist of κ positions of bits that are one in the polynomial $\mathbf{c}$, the Gaussian distributed polynomial $\mathbf{z}_1$ (std. deviation σ), and the small polynomial $\mathbf{z}_2^\dagger$ where most lower order bits have already been dropped. Storing the signature in this format would require $\approx \kappa \cdot \log_2(n) + n \cdot \lceil (1 + \log_2(\tau\sigma)) \rceil + n \cdot \lceil (\log_2(3) \rceil$ bits, which is $\approx$ 8,399 bits for BLISS-I. However, in order to achieve the signature size stated in Table 3.3 of 5,600 bits for BLISS-I additional Huffman encoding of $\mathbf{z}_1, \mathbf{z}_2^\dagger$ is necessary (c does not contain any easily removable redundancy). Savings are possible as small coefficients are much more likely than large ones in a polynomial distributed according to a discrete Gaussian distribution.

In order to perform the Huffman encoding efficiently, we aim at getting a small Huffman table by not encoding too many values. Therefore, we focus on the higher-order bits of coefficients of $\mathbf{z}_1, \mathbf{z}_2$ (for $\mathbf{z}_2$ already given by $\mathbf{z}_2^\dagger$), since the lower order bits are almost uniform and thus not efficiently compressible. As the distributions are symmetric, we encode only absolute values, and deal with the signs separately which further reduces the table size. To decrease the overhead between theoretical entropy and actual efficiency of Huffman encoding we group two coefficients from $\mathbf{z}_1$ and two coefficients from $\mathbf{z}_2^\dagger$ together.

To encode a signature we thus split $\mathbf{z}_1, \mathbf{z}_2^\dagger$ into $n/2$ blocks of the form $b[i] = (\mathbf{z}_1[2i], \mathbf{z}_1[2i + 1], \mathbf{z}_2^\dagger[2i], \mathbf{z}_2^\dagger[2i+1])$ and focus from now on a single block $b = (z_1, z_1', z_2, z_2')$. The components z_1, and z_1', respectively, are then decomposed as a triple of higher-order bits h_{z_1}, lower-order bits l_{z_1}, and sign $s_{z_1} \in \{-1, 0, 1\}$ where $z_1 = s_{z_1}(\cdot h_{z_1} \cdot B + l_{z_1})$ with $B = 2^\beta$ and $l_{z_1} \in [0, \ldots, B-1]$. Note that the value of s_{z_1} is irrelevant in case the decomposed value is zero and that the coefficients from $\mathbf{z}_2$ already have their lower-order bits dropped (thus $h_{z_2} = z_2$ and $h_{z_2'} = z_2'$). For possible values of $(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$ we have calculated the frequencies and accordingly a variable length Huffman encoding where $\tilde{e} = \textsc{encode}(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$. The sign bits are also stored as a variable length $\tilde{s}$ string where sign bits are only stored if the associated coefficient is non zero

Table 7.5: Huffman table for signature compression (BLISS-I parameter set).

$(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$	Binary string	$(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$	Binary string
(0, 0, 0, 0)	100	(2, 0, 0, 0)	00011
(0, 0, 0, 1)	01000	(2, 0, 0, 1)	0000111
(0, 0, 1, 0)	01001	(2, 0, 1, 0)	0000101
(0, 0, 1, 1)	0011100	(2, 0, 1, 1)	000001001
(0, 1, 0, 0)	110	(2, 1, 0, 0)	00110
(0, 1, 0, 1)	01101	(2, 1, 0, 1)	0001000
(0, 1, 1, 0)	01011	(2, 1, 1, 0)	0001011
(0, 1, 1, 1)	0011110	(2, 1, 1, 1)	000001101
(0, 2, 0, 0)	00100	(2, 2, 0, 0)	0000001
(0, 2, 0, 1)	0000100	(2, 2, 0, 1)	0000011111
(0, 2, 1, 0)	0000110	(2, 2, 1, 0)	000000000
(0, 2, 1, 1)	000001010	(2, 2, 1, 1)	000001111001
(0, 3, 0, 0)	000000011	(2, 3, 0, 0)	000001111000
(0, 3, 0, 1)	00000000101	(2, 3, 0, 1)	00000001011010
(0, 3, 1, 0)	000001111011	(2, 3, 1, 0)	00000001011000
(0, 3, 1, 1)	00000111101000	(2, 3, 1, 1)	0000011110101 1010
(1, 0, 0, 0)	101	(3, 0, 0, 0)	000001000
(1, 0, 0, 1)	01100	(3, 0, 0, 1)	00000000100
(1, 0, 1, 0)	01010	(3, 0, 1, 0)	00000000110
(1, 0, 1, 1)	0011101	(3, 0, 1, 1)	00000001011011
(1, 1, 0, 0)	111	(3, 1, 0, 0)	000001011
(1, 1, 0, 1)	01110	(3, 1, 0, 1)	00000001010
(1, 1, 1, 0)	01111	(3, 1, 1, 0)	00000000111
(1, 1, 1, 1)	0011111	(3, 1, 1, 1)	00000111101001
(1, 2, 0, 0)	00101	(3, 2, 0, 0)	000000010111
(1, 2, 0, 1)	0001001	(3, 2, 0, 1)	00000001011001
(1, 2, 1, 0)	0001010	(3, 2, 1, 0)	000001111010111
(1, 2, 1, 1)	000001110	(3, 2, 1, 1)	0000011110101 1011
(1, 3, 0, 0)	000001100	(3, 3, 0, 0)	0000011110101 1001
(1, 3, 0, 1)	00000001000	(3, 3, 0, 1)	000001111010110000
(1, 3, 1, 0)	00000001001	(3, 3, 1, 0)	0000011110101100011
(1, 3, 1, 1)	00000111101010	(3, 3, 1, 1)	0000011110101100010

Computed for standard deviation $\sigma \approx 215$, with $B = 2^\beta$ for $\beta = 8$.

(maximum four bits). As a consequence, the encoding of a whole block $b[i]$ is the concatenation $v = \tilde{s}|\tilde{e}|l_{z'_1}|l_{z_1}$. During decoding the values of $(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$ have to be recovered from $\tilde{e}$ and (z_1, z'_1, z_2, z'_2) can be computed using $l_{z'_1}, l_{z_1}$ and the sign information.

For our FPGA implementation we have realized a separate encoder `HuffmanEncode` and decoder `HuffmanDecode` component exclusively for the BLISS-I parameter set and developed wrappers `SignHuff` and `VerifyHuff` using them on top of `Sign` and `Verify`, respectively (see Figure 7.5). For faster development time we rely on high-level synthesis (HLS) to implement both cores[5]. The encoder requests pairs of $\mathbf{z}_1, \mathbf{z}_2^\dagger$ from a small FIFO necessary for short term buffering and because $\mathbf{z}_2^\dagger$ coefficients are slightly more delayed due to compression. The $\tilde{e} = \textsc{Encode}(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$ function is realized as a look-up table where the concatenated value $h_{z_1}|h_{z'_1}|h_{z_2}|h_{z'_2}$ is used as an address for a table with 64 entries (see Table 7.5). The final encoded signature is then written to the top-level component in 32-bit chunks. The maximum size of one v is 39 bits with a maximum length of $\tilde{e}$ of 19 bits, 2 times 8 bits for the $l_{z'_1}|l_{z_1}$ and 4 bits for signs. It can happen that from a previous run at maximum 31 bits stay in the internal buffer of the encoder (with 32 or more bits the buffer would have been written to the output). For decoding we first request chunks of the encoded signature into a shift register. The values of $(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$ for a given e are recovered by linear searching in the Huffman table ordered by the size/probability of resulting bit strings. Using this information the signature can be completely recovered and is stored in an internal dual-block memory instantiated by the `HuffmanDecode` component. In the `VerifyHuff` top-level component this buffer is connected to the `Verify` component and the buffer allows parallel decoding of one signature and verification of an already decoded signature after it has been read from the `HuffmanDecode` buffer.

7.3.4 Results

In this section we discuss our results which were obtained post-PAR on a Spartan-6 LX25 (speed grade -3) with Xilinx ISE 14.7.

Gaussian Sampling

Detailed results on area consumption and timing of our two discrete Gaussian sampler designs (`SamplerBER` and `SamplerCDT`) are given in Table 7.7. The results show that the enhanced CDT sampler consumes less logic resources than the Bernoulli sampler at the cost of one 18k block memory to store the table B. This is a significant improvement in terms of storage size compared to a naive implementation without the application of the Kullback-Leibler divergence and Gaussian convolution. A standard CDT sampler implementation would require at least $\sigma\tau\lambda = 370$ kbits (that is about 23 18K block RAMs) for the defined parameters matching a standard deviation $\sigma = 215.73$, tailcut $\tau = 13.4$, and precision $\lambda = 128$.

---

[5] While a hand-optimized plain VHDL implementation would probably be more efficient, we opted for the HLS design flow mainly due to much higher development speed and faster verification using a C testbench. As Huffman encoding is not a core component of the signature scheme and not a particularly new technique it did not seem worthwhile to spend a large amount of time with low-level design of such a component. However, in order to provide a complete implementation that achieves the theoretical signature size, Huffman encoding is required and by using a HLS tool we can give good estimates for resource consumption and runtime (or at least an upper bound). However, in future work it would certainly be interesting to compare our implementation to a hand-optimized low-level VHDL implementation.

Regarding randomness consumption the CDT sampler needs on average 21 bits for one sample (using two smaller samples and the convolution theorem) which are generated by three instantiations of Trivium. The Bernoulli sampler on the other hand consumes 33 bits on average where 12% of the random bits are consumed by the $\mathtt{D}_{\sigma_{\mathtt{bin}}}$ module, 42% by the uniform sampler, 43% by both $\mathtt{B}_{\mathtt{exp(-x/f)}}$ units and 2.8% for the final sign determination and zero rejection. As illustrated in Figure 7.4, we feed the Bernoulli sampler with the pseudo-random output of two Trivium instances and a significant amount of the logic consumption can be attributed to additional buffers to compensate for possible rejections and distribution of random bits to various modules. With respect to the averaged performance, 7.5 and 17.95 cycles are required by the CDT and the Bernoulli sampler to provide one sample, respectively. We also provide results for instantiations of the CDT sampler for larger standard deviations required by BLISS-III and BLISS-IV that show that the performance impact caused by the increased standard deviation σ is small. In general, the `SamplerBER` component could also be instantiated for larger standard deviations but in this case random number generation and distribution and thus the design would have to be changed for a fair comparison (e.g., sampling of k in Algorithm 5 requires more random bits for $\sigma > 217.34$). As a consequence, we just give results for the optimized and balanced instantiation with $\sigma \approx 215$.

With regard to the `SamplerCDT` component, by combining the convolution lemma and Kullback-Leibler divergence we were able to maintain the advantage of the CDT, namely high speed and relative simple implementation, but significantly reduced the memory requirements (from $\approx$ 23 18K block RAMs to one 18K block RAM). The convolution lemma works especially well in combination with the reverse tables as the overall table sizes shrink and thus the number of comparisons is reduced. Thus, we do not expect a CDT sampler that samples directly from standard deviation σ to be significantly faster. Additionally, larger tables would require more complex address generation which might lower the achievable clock frequency. The Bernoulli approach on the other hand does not seem as suitable for an application of the convolution lemma as the CDT. The reason is that the tables are already very small and thus a reduction would not significantly reduce the area usage. Moreover, sampling from the binary Gaussian distribution σ_{bin} ($\mathtt{D}^{+}_{\sigma_{\text{bin}}}$ component) is independent of the target distribution and does not profit from a smaller σ.

Previous implementations of Gaussian sampling for lattice-based public-key encryption can be found in [RVV13, PG14]. However, both works target a smaller standard deviation of $\sigma = 3.3$. The work of Roy et al. [RVV13] uses the Knuth-Yao algorithm (see [DG14] for more details), is very area-efficient (47 slices on a Virtex-5), and consumes few randomness but requires 17 clock cycles for one sample. Bernoulli sampling can also be used to optimize simple rejection sampling by using Bernoulli evaluation instead of computation of exp() (see Section 5.4). However, without usage of the binary Gaussian distribution (see [DDLL13a]) the rejection rate is high and one sample requires 96 random bits and 144 cycles. This is acceptable for a relatively slow encryption scheme and possible due to the high output rate (one bit per cycle) of the used stream cipher but not a suitable architecture for BLISS. The discrete Ziggurat [BCG+13] performs well in software and might also profit from the techniques introduced in this work but does not seem to be a good target for a hardware implementation due to its infrequent rejection sampling operations and its costly requirement of high-precision floating point arithmetic.

Table 7.7: Performance and resource consumption of our implementation of various BLISS parameter sets on reconfigurable hardware.

Configuration and Operation	Slices	LUT FF	BRAM DSP	MHz	Cycles	Operations per second (output)
`SignHuff` (BLISS-I, CDT)	2,291	7,193/6,420	5.5/5	139	≈15,864	8,761 (signature)
`VerifyHuff` (BLISS-I)	1,687	5,065/4,312	4/3	166	≈16,346	17,101* (valid/invalid)
`Sign` (BLISS-I, 2×BER)	2,646	8,313/7,932	5/7	142	≈15,840	≈8,964 (signature)
`Sign` (BLISS-I, CDT)	2,047	6,309/6,127	6/5	140	≈15,864	≈8,825 (signature)
`Sign` (BLISS-III, CDT)	2,079	6,397/6,179	6.5/5	133	≈27,547	≈4,828 (signature)
`Sign` (BLISS-IV, CDT)	2,141	6,438/6,198	7/5	135	≈47,528	≈2,840 (signature)
`Verify` (BLISS-I)	1,482	4,366/3,887	3/3	172	9,607	17,903 (valid/invalid)
`Verify` (BLISS-III)	1,435	4,298/3,867	3/3	172	9,628	17,760 (valid/invalid)
`Verify` (BLISS-IV)	1,399	4,356/3,886	3/3	171	9,658	17,809 (valid/invalid)
`SamplerBER` (BLISS-I)	452	1,269/1,231	0/1	137	≈17.95	≈7,632,311 (sample)
`SamplerCDT` (BLISS-I)	299	928/1,121	1/0	129	≈7.5	≈17,100,00 (sample)
`SamplerCDT` (BLISS-III)	265	880/1,122	1.5/0	133	≈7.56	≈17,592,593 (sample)
`SamplerCDT` (BLISS-IV)	281	922/1,123	2/0	133	≈7.78	≈17,095,116 (sample)
`PolyMul` (CDT, BLISS-I)	835	2,557/2,707	4.5/1	145	9,307	15,579 ($\mathbf{a} \cdot \mathbf{y}_1$)
`Butterfly`	127	410/213	0/1	195	6	$195 \cdot 10^6$ [pipelined]
`Hash` ($Nb = 16$)	752	2,461/2,134	0/0	149	1,931	77,162 ($\mathbf{c}$)
`SparseMul` ($C = 1$)	64	162/125	0/0	274	15,876	17,258 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`SparseMul` ($C = 8$)	308	918/459	0/0	267	2,436	109,605 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`SparseMul` ($C = 16$)	628	1,847/810	0/0	254	1,476	172,086 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`Compression`**	232	700/626	0/4	150	-	parallel to `SparseMul`
`EncodeHuff` (BLISS-I)	244	752/244	0/0	150	-	parallel to `Sign`
`DecodeHuff` (BLISS-I)	259	795/398	0/0	159	≈5,639	28,196 ($\mathbf{z}_1, \mathbf{z}_2^\dagger$)

By ≈ we denote averaged cycle counts. The post PAR results where synthesized on a Spartan-6 LX25-3 for a 1024 bit message. (*) Regarding throughput the cycle count of the Huffman enabled verification core is equal to the standard core as the decoded signature is saved in a buffer RAM and thus the decoding and verification can work in parallel. However, the latency of `VerifyHuff` is $Cycles($`Verify`$)+$ $Cycles($`DecodeHuff`$)$. (**) In the conference version `Compression` also contained the area count of the `Hash` and `SparseMul` components. However, this does not match the block diagram in Figure 7.5.

BLISS Operations

Results for our implementation of the BLISS signing and verification engine and sub-modules can be found in Table 7.7 including averaged cycle counts and possible operations per second (sometimes considering pipelining).[6] The `SignHuff` and `VerifyHuff` top-level modules include the Huffman encoding for very short signatures which is just implemented for the BLISS-I parameter set. The impact of Huffman encoding on the signing performance is negligible as the encoding is performed in parallel to the signing process. For verification we save the decoded signature obtained from the `DecodeHuff` component in a buffer which is then accessed by the verification core. As a consequence, the latency of the verification operation is $\text{Cycles}(\texttt{Verify})+\text{Cycles}(\texttt{DecodeHuff})$ but for high throughput a signature can be decoded while another signature is verified. Thus the number of verification operations per second is not affected and similar to the amount of operations possible without Huffman decoding. For BLISS-I one signing attempt takes roughly 10,000 cycles and on average 1.6 trials are necessary using the BLISS-I parameter set. To evaluate the impact of the sampler used in the design, we instantiated two signing engines (`Sign`) of which one employs a CDT sampler and the other one two Bernoulli samplers to match the speed of the multiplier. For a similar performance of roughly 9,000 signing operations per second, the signing instance based on the Bernoulli sampler has a higher resource consumption (about 600 extra slices). Due to the two pipeline stages involved, the runtime of both instances is determined by $\max(\text{Cycles}(\texttt{PolyMul}), \text{Cycles}(\texttt{Hash})) + \text{Cycles}(\texttt{SparseMul})$ where the rejection sampling in `Compression` is performed in parallel. Further design space exploration, e.g., evaluating the impact of a different number of parallel sparse multiplication operations or a faster configuration of KECCAK always identified the `PolyMul` component as performance bottleneck or did not provide significant savings in resources for reduced versions. In order to further increase the clock rate it would of course also be possible to instantiate the Gaussian sampler in a separate clock domain. The verification runtime is determined by $\text{Cycles}(\texttt{PolyMul}) + \text{Cycles}(\texttt{Hash})$ as no pipelining is used inside of `Verify`. The `PolyMul` is slightly faster than for signing as no Gaussian sampling is needed. It is also worth noting that for higher security parameter sets like BLISS-III (160-bit security) and BLISS-IV (192-bit security) the resource consumption does not increase significantly. Only the signing performance suffers due to a higher repetition rate (see Table 3.3). Verification speed and area consumption are almost constant for all security levels.

7.4 Comparison of our Implementations with Related Work

In Table 7.9 we summarize our results and provide implementation results for other signature schemes. The main difference of our work, compared to the first implementation of the GLP scheme [GLP12], is the use of the NTT instead of a schoolbook approach for polynomial multiplications (i.e., for computation of $\mathbf{a}\mathbf{y}_1+\mathbf{y}_2$). The schoolbook multiplier of [GLP12] was realized

---

[6] We also give size, runtime, and achievable clock frequency estimates for sub modules. However, due to cross-module optimization, different design strategies, and constraint options the resource consumption cannot just be added and varies slightly (e.g., achievable timing of a stand-alone instantiation might be lower than when instantiated by a top-level module). Moreover, the target clock frequency in the constraints file can heavily influence the achievable clock frequency (if too low, PAR optimization stops early, if too high, PAR optimization gives up too quickly), and while we tried to find a good configuration for the top level modules we just synthesized the sub modules as they are with a generic constraints file.

using a DSP and placed in a separate clock domain to achieve high frequencies (e.g., 416 MHz on a Virtex-6). However, due to the need to perform roughly $n^2 = 512^2 = 262144$ multiplications in $\mathbb{Z}_q$ to compute $\mathbf{ay}_1$ and despite the high operating frequency several multipliers were required to saturate the other parts of the signing engine. Thus the number of required DSP blocks is rather high in [GLP12] (i.e., 28 instead of the four used in our work). Another difference is that we now compute $\mathbf{z}_1, \mathbf{z}_2$ in parallel and not sequentially as in [GLP12]. Furthermore, and contrary to [GLP12], we remark that our implementations of GLP and BLISS take the area costs and timings of a hash function into account.

For a fair comparison with [GLP12] we compare only the separate signing and verification cores. However, our BOTH core provides support for both operations at almost no additional costs compared to a core supporting only the signing operation. On the Spartan-6 platform we need 75% LUTs, 92% BRAMs and 14 % DSPs for signing (see Table 7.3) compared with the results of [GLP12] given in Table 7.9. With respect to performance, the cores presented in this thesis can sign 1715 signatures per second compared to the previous 931 which is an improvement of a factor of 1.8. Since in [GLP12] the computation of $\mathbf{az}_1 + \mathbf{z}_2' - \mathbf{tc}$ is not performed in parallel with just one schoolbook multiplier, the performance for verification of this work is even better, due to the usage of the NTT. Our verification core consumes almost the same amount of BRAMs but only 67% LUTs, and 14 % DSPs and increases the performance by a factor of 7.4.

When just comparing the GLP and BLISS implementations given in this thesis, it becomes evident that the BLISS design achieves higher throughput with a similar number of DSPs and logic resources but only a fourth of the block memories required by GLP. The main reasons for this advantage are that BLISS operates with a smaller modulus (GLP: $q = 8383489$/BLISS-I: $q = 12289$) and that it requires less iterations to produce a valid signature (GLP: 7/BLISS-I: 1.6). Moreover, BLISS provides a higher security level (GLP: 80 bit/BLISS-I: 128 bit) than GLP. The biggest advantage of our GLP implementation (besides the improvements compared to [GLP12]) is that one core is a signing/verification hybrid which might be useful for some application scenarios and that an implementation of key generation could be added easily. A signing/verification hybrid core (even supporting multiple security levels) would also be possible for BLISS but would require a significant additional engineering and testing effort. However, an implementation of BLISS key generation appears to be more challenging than GLP key generation due to the more complicated rejection sampling and computation of an inverse. In summary, our implementation of BLISS is superior to [GLP12] and also our GLP implementation in almost all aspects.

Compared with implementations of RSA and ECC the performance and area consumption of our implementations is competitive or even better. As an example, Glas et al. [GSS+11] report a vehicle-to-X communication accelerator based on an ECDSA signature over 256-bit prime fields. With respect to their work, our BLISS implementation shows higher performance at less resource cost. An ECDSA implementation on a binary curve for an 80-bit security level on an Altera FPGA is given in [JS07] and achieves similar speeds and area consumption compared to our work. Other ECC implementations over 256-bit prime or binary fields (e.g., such as [GP08] on a Xilinx Virtex-4) only implement the point multiplication operation and not the full ECDSA protocol. Finally, a fast RSA-1024/2048 hybrid core was presented for Virtex-4 devices in [SM11] which requires more logic/DSPs and provides significantly lower performance (12.6 ms per 2048-bit private key operation) than our core.

For the NTRUSign lattice-based signature scheme (introduced in [HHP$^+$03] and broken by Nguyen [NR09]) and the XMSS [BDH11] hash-based signature scheme we are not aware of any implementation results for FPGAs. Implementations of several post-quantum multivariate quadratic ($\mathcal{MQ}$) signature schemes like Unbalanced Oil and Vinegar (UOV), Rainbow, and TTS were given in [BERW08]. These schemes are usually faster (factor 2-50) than ECC but also suffer from large key sizes for the private and public key (e.g., 80 Kb for UOV) [PTBW11]. While implementations of the McEliece encryption scheme offer good performance [EGHP09,SWM$^+$10] the only implementation of a code based signature scheme [BSTV04] is extremely slow with a runtime of 830 ms for signing.

7.5 Conclusion and Future Work

With this work we have shown that lattice-based digital signature schemes that support a broad range of security levels (80-bit to 192-bit security) can be implemented efficiently on low-cost FPGAs. Moreover, we have given an implementation of an efficient and theoretically sound discrete Gaussian sampler using a small table. Our work on BLISS shows that Gaussian sampling can be implemented efficiently on FPGAs and that the advantages of using Gaussian noise (e.g., security, smaller signatures, less rejections) outweigh the added resource costs compared to schemes relying on uniform noise.

For future work it seems worthwhile to investigate the properties of other samplers (e.g., Knuth-Yao [RVV13, DG14]) and to implement different signature schemes like NTRUSign with secure rejection sampling [MBDG14] or PASSSign [HPS$^+$14]. Moreover, for practical adoption of BLISS protection against side-channels is required. By using new compression techniques or other tricks it might also be possible to further reduce the size of the BLISS signature or to increase performance (see [Duc14]). For practical applications an (open-source) BLISS core supporting multiple security levels and signing as well as verification might be useful. However, for most real world applications a core would require protection against side-channel attacks and fault injection. In general, also other configuration options could be explored (e.g., more pipelining stages for the high-repetition rate BLISS-IV parameter set), usage of different hash functions, or a faster NTT multiplier. As our work presented in this thesis focuses on speed it is also not clear how small and fast a lightweight implementation of BLISS would be.

Table 7.9: Signing and verification performance of our FPGA implementation of GLP and BLISS in comparison with implementations of other signature schemes.

Operation	Security	Device	Resources	Ops/s
BLISS-I, **our work** [`SignHuff`]	128	XC6SLX25	7,193 LUT/ 6,420 FF 5 DSP/ 5.5 BRAM18	8,761
BLISS-I, **our work** [`VerHuff`]	128	XC6SLX25	5,065 LUT/4,312 FF 3 DSP/ 4 BRAM18	17,101
GLP-I, **our work** [`Sign`/`Verify`]	80	XC6SLX25	6,088 LUT/ 6,804 FF/ 4 DSP/ 19.5 BRAM18	1,627/ 7,438
BLISS-I, conf. version [`Sign`] [PDG14a]	128	XC6SLX25	7,491 LUT/7,033 FF 6 DSP/ 7.5 BRAM18	7,958
BLISS-I, conf. version [`Verify`] [PDG14a]	128	XC6SLX25	5,275 LUT/1,727 FF 3 DSP/ 4.5 BRAM18	14,438
GLP-I [sign] [GLP12]	80	XC6SLX16	7,465 LUT/ 8,993 FF/ 28 DSP/ 29.5 BRAM18	931
GLP-I [verify] [GLP12]	80	XC6SLX16	6,225 LUT/ 6,663 FF/ 8 DSP/ 15 BRAM18	998
ECDSA-secp256r1 [sign/ver] [GSS+11]	128 128	XC5VLX110T XC5VLX110T	32,299 LUT/FF pairs 32,299 LUT/FF pairs	139 110
ECDSA-B163 [sign/ver] [JS07]	80	EP2C20	15,879 LE / 8,472 FF/ 36 M4K	1,063/ 621
RSA-1024/2048 [sign] [SM11]	80/103	XC4VFX12-10	4190 SLICES/17 DSP 7 BRAM18	548/ 79
ECDSA-B163 [point mult.]* [AH08]	80	XC4VLX200	7,719 LUT/ 1,502 FF	47,619
ECDSA-P224 [point mult.]* [GP08]	112	XC4VFX12-12	1,580 LS/ 26 DSP	2,739
ECDSA-P256* [point mult.] [MLPJ13]	128	XC5LX100	4,177 LUT/ 4,792 FF 37 DSP/ 18 BRAM36	2,631
ECDSA-25519 [point mult.]* [SG14]	128	XC7Z020	2,783 LUT/ 3,592 FF 20 DSP/ 2 BRAM36	2,518

(*) The overall cost of an ECDSA signing operation is dominated by one point multiplication but a full core would also require a hash function, logic for arithmetic operations, and inversion. ECDSA verification requires one or two point multiplications depending on the curve representation but also a hash function and logic for arithmetic operations.

Chapter 8

Implementation of Lattice-Based Signatures in Software

In this chapter we examine the software performance of lattice-based signature schemes on several platforms. We first revisit the vectorized GLP *implementation from [GOPS13] and show two simple optimizations for further speed-up. The second contribution is an implementation of* BLISS *on a Cortex-M4F device and the evaluation of several discrete Gaussian samplers. Moreover, we provide an implementation of* BLISS *on the ATxmega which relies on a cumulative distribution table (CDT) sampler combined with convolutions of Gaussians (see Section 2.5.4) and previously discussed improvements of the NTT for software platforms (see Section 6.2). The optimization of* GLP *proposed in this work appeared in the full version of [DBG⁺14]. The implementation of* BLISS *on the Cortex-M4F has been published in [OPG14] and some of the results of the paper are based on the Bachelor's thesis of Tobias Oder [Ode13]. The* BLISS *implementation on the ATxmega platform appeared in [POG15a]. Moreover, some content from [HPO⁺15] is included and we refer to results published in works like [GOPS13, DBG⁺14] that have been co-authored by the author of this thesis.*

Contents of this Chapter

8.1 Introduction

If predictions regarding the upcoming Internet-of-things or Industry 4.0 hold, more and more energy and cost-constrained embedded systems will be deployed in decentralized networks that are connected to massive cloud services [AIM10, GBMP13]. To authenticate communicating parties or software updates, long-term secure digital signature schemes (DSS) are required that can be implemented efficiently on small devices but that also support fast execution on servers. Candidate schemes could be the recently proposed lattice-based digital signatures like GLP and BLISS, which are also fast in hardware (see Chapter 7). A particular advantage of these schemes is that they rely on polynomial multiplication as core function. Thus, no complicated data structures or multiprecision arithmetic is necessary for their implementation. Most parts of the computation basically require a large number of operations in $\mathbb{Z}_q$, usually for $\lceil \log_2(q) \rceil \leq 32$ that can be computed rather efficiently on common 8-bit, 16-bit, or 32-bit microcontrollers. Additionally, for server implementation it has been shown in [GOPS13] that the NTT can be vectorized efficiently and even a plain C implementation of BLISS already achieves high speed [DDLL13a]. However, Gaussian sampling and polynomial multiplication are still expensive and some effort is required to make implementations fast on a wide range of microcontrollers and microprocessors.

8.1.1 Related Work

The GLP signature scheme was introduced in [GLP12] (see Section 3.5) and BLISS was proposed in [DDLL13a] (see Section 3.6). Hardware implementations of GLP and BLISS are covered in Chapter 7. A vectorized software implementation of GLP has been proposed in [GOPS13] and a proof of concept software implementation of BLISS is already provided with the original paper [DDLL13a]. An implementation of PASSSign is described in [HPS+14]. A library (NFLlib) for the implementation of ideal lattice-based cryptography using the NTT and the Chinese remainder theorem (CRT) is described in [MBFK14]. In [BSJ14, BJ14] implementations of BLISS and GLP (instantiated as identification scheme) on 8-bit AVR platforms are proposed. An optimized implementation of the NTT on the Cortex-M4F that improves upon the results discussed in this chapter is provided by de Clercq et al. in [dCRVV15]. Usually, real world instantiations of lattice-based signature schemes rely on ideal lattices to reduce the size of public and private keys. One exception is an implementation of BG signatures [BG14] provided in [DBG+14] that shows that even an instantiation using standard lattices can achieve high performance in software.

8.1.2 Contribution

We revisit the implementation of GLP provided in [GOPS13] and show high-level optimizations that increase signing speed by a factor of 1.4 without modification of the underlying vectorized NTT implementation. Moreover, we discuss a possible approach for the implementation of vectorized sparse multiplication. While this approach does not lead to performance gains compared to the NTT, it might be applicable for alternative schemes or on different architectures.

Additionally, we present an implementation of the BLISS signature scheme tailored to a 32-bit ARM Cortex-M4F RISC (1024 KB flash/192 KB SRAM) microcontroller capable of running with up-to 168 MHz. This common series of ARM microcontrollers is not only deployed in vehicular environments but also in many other embedded devices, such as smart meter gateways, medical devices, or industrial control systems so that our results are also likewise applicable to many other applications. For this platform, we investigate the optimal implementation of polynomial multiplication using the NTT as well as an analysis of the efficiency of several Gaussian samplers. Our implementation on this low-cost platform achieves a significant performance, namely 28 signing and 167 verification operations per second, outperforming classical cryptosystems such as RSA and ECC.

Targeting 8-bit architectures we provide the, up to our knowledge, fastest and smallest implementation of BLISS on the AVR platform. We show the benefits of using the optimized

NTT algorithms from Chapter 6, and the approach of using the CDT and convolutions that has previously been successfully implemented in hardware (see Section 7.3.1). One signature computation requires only 329 ms and verification requires 88 ms.

8.2 Improved Implementation of GLP on Intel/AMD CPUs

In [GOPS13] Güneysu et al. described an optimized software implementation of the GLP signature scheme. The implementation for Intel's Sandy Bridge and Ivy Bridge in particular targets the advanced vector extensions (AVX) providing support for single instruction multiple data (SIMD) operations. Their C-implementation features storing of parameters in NTT representation, lazy reduction, and representation of 512-coefficient polynomials as a 512 double-precision array of floating-point values. By utilizing the AVX instruction set that implementation can perform up to four multiplications and four additions of coefficients in each cycle. The reported (average) cycle count for a successful signing operation is 634,988 cycles, while verification takes 45,036 cycles and key generation 31,140 cycles.

However, in [GOPS13] the secret key $(\mathbf{s}_1, \mathbf{s}_2)$ is not stored in the NTT domain and transformed during every signing attempt. This motivated a closer look at the implementation and led to some optimizations and an alternative method for sparse multiplication.

8.2.1 Faster Uniform Sampling and Better Exploitation of the NTT

As mentioned, in [GOPS13] the secret key $\mathsf{sk} = (\mathbf{s}_1, \mathbf{s}_2)$ is not stored in NTT representation and can thus be compressed into only 256 bytes. This is possible as $\mathbf{s}_1$ and $\mathbf{s}_2$ are polynomials in $\mathcal{R}_q$ with $n = 512$ coefficients that are chosen uniformly random from $\{-1, 0, 1\}$. However, when a larger secret key is acceptable, $\mathbf{s}_1$ and $\mathbf{s}_2$ can be stored in NTT representation in which a polynomial requires at least $n\lceil \log_2 q \rceil$ bits ($q = 8383489$ in parameter set I). Thus, no forward transforms of $\mathbf{s}_1$ and $\mathbf{s}_2$ are required during signing. As a consequence, in our improved implementation the computation of $\mathbf{cs}_1$ and $\mathbf{cs}_2$ can be performed with one forward transformation of $\mathbf{c}$, two point-wise multiplications with the already transformed $\mathbf{s}_1$ and $\mathbf{s}_2$, and two inverse transforms. The secret key size of our optimized version is $2 \cdot 512 \cdot 8 =$ 8,192 bytes where each coefficient is being stored in floating point representation as a 64-bit double variable.

Additionally, we optimized the uniformly random sampling of $\mathbf{y}_1$ and $\mathbf{y}_2$. GLP signing requires to sample $\mathbf{y}_1 \xleftarrow{\$} [-k, k]^n$ and $\mathbf{y}_2 \xleftarrow{\$} [-k, k]^n$ for $k = 2^{14}$ and $n = 512$ (parameter set I). In [GOPS13] a pool of 2,112 random bytes is sampled using a stream cipher and (with a high probability) a 32-bit random integer results in a random coefficient of $\mathbf{y}_1$ or $\mathbf{y}_2$ that is in $[-2^{14}, 2^{14}]$. However, it is easy to see that there is a considerable loss of entropy by converting 32 bits into a 16-bit value. The whole sampling process can be significantly improved when the original parameter set I is changed[1] and k is set to $k = 2^{14} - 1$. We can now use the same approach to sample as described in [DBG+14] and only need 1,056 pseudo random bytes for a signing attempt.

The verification speed can also be improved, again by using the NTT. In Section 7.2.2 it was proposed to compute the input $\mathbf{az}_1 + \mathbf{z}_2' - \mathbf{tc}$ to the random oracle during verification as $\mathsf{INTT}(\tilde{\mathbf{a}} \circ \mathsf{NTT}(\mathbf{z}_1) - \tilde{\mathbf{t}} \circ \mathsf{NTT}(\mathbf{c})) + \mathbf{z}_2'$ where $\mathsf{pk} = (\tilde{\mathbf{a}}, \tilde{\mathbf{t}})$ is stored in NTT representation. The

---

[1] As we consistently reduce k only by one the impact on the correctness and security of the original scheme is negligible.

addition of two polynomials in NTT representation is possible due to the linearity of the NTT and thus only one inverse transformation, instead of two separate inverse transformations of $\tilde{\mathbf{a}} \circ \mathsf{NTT}(\mathbf{z}_1)$ and $\tilde{\mathbf{t}} \circ \mathsf{NTT}(\mathbf{c})$, is necessary.

8.2.2 Notes on Vectorized Sparse Multiplication for GLP

We also developed an alternative approach for the efficient and vectorized implementation of sparse multiplication without using the NTT. However, for the computation of $\mathbf{cs}_1$ and $\mathbf{cs}_2$ we were not able to achieve better performance or a smaller secret key than with the NTT. While our approach did not result in a speed-up on the current architecture it might be advantageous on different architectures. Our idea is based on the previously mentioned fact that in GLP the output $\mathbf{c}$ of the random oracle is a sparse polynomial of weight 32 which is multiplied by polynomials $\mathbf{s}_1$ and $\mathbf{s}_2$ with small coefficients chosen uniformly from $\{-1, 0, 1\}$. The application of a standard row-wise multiplication algorithm to this problem would usually require the addition of $n = 512$ shifted coefficients of $\mathbf{s}_1$ and $\mathbf{s}_2$ to the final results for every coefficient that is one or minus one in $\mathbf{c}$. However, usage of this approach (even with vectorization) is not competitive compared to the NTT as it still requires $32 \cdot 512 = 16{,}384$ additions or subtractions.

However, it is easy to see that the coefficients of a result of a multiplication $\mathbf{h}_1 = \mathbf{cs}_1$ and $\mathbf{h}_2 = \mathbf{cs}_2$ are small (maximum $32n$) and not even remotely fill the 53-bit mantissa of a double register. As a consequence, we pack seven coefficients of $\mathbf{s}_1$ and $\mathbf{s}_2$ into one double value. This is possible as we need at most seven bits per coefficient and can thus prevent an overflow inside the double value when computing $\mathbf{cs}_1$ and $\mathbf{cs}_2$. Basically, this now allows us to add or subtract 28 coefficients with one 256-bit AVX vector operation working on four double precision floating point values at the same time. However, now the shifts resulting from the position of a one in $\mathbf{c}$ are even more complicated, as we would also have to shift and negate the seven coefficients packed into one double. To solve this issue, we store seven rotations of $\mathbf{s}_1$ and $\mathbf{s}_2$, respectively. The first and the n-th double element of each rotation of either $\mathbf{s}_1$ or $\mathbf{s}_2$ are shown exemplary:

$$\left|\begin{array}{ccccccc}
-s_0 & -s_1 & -s_2 & -s_3 & -s_4 & -s_5 & -s_6 \\
0 & -s_0 & -s_1 & -s_2 & -s_3 & -s_4 & -s_5 \\
0 & 0 & -s_0 & -s_1 & -s_2 & -s_3 & -s_4 \\
0 & 0 & 0 & -s_0 & -s_1 & -s_2 & -s_3 \\
0 & 0 & 0 & 0 & -s_0 & -s_1 & -s_2 \\
0 & 0 & 0 & 0 & 0 & -s_0 & -s_1 \\
0 & 0 & 0 & 0 & 0 & 0 & -s_0
\end{array}\right|$$

$$\left|\begin{array}{ccccccc}
s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\
-s_{n-1} & s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\
-s_{n-2} & -s_{n-1} & s_0 & s_1 & s_2 & s_3 & s_4 \\
-s_{n-3} & -s_{n-2} & -s_{n-1} & s_0 & s_1 & s_2 & s_3 \\
-s_{n-4} & -s_{n-3} & -s_{n-2} & -s_{n-1} & s_0 & s_1 & s_2 \\
-s_{n-5} & -s_{n-4} & -s_{n-3} & -s_{n-2} & -s_{n-1} & s_0 & s_1 \\
-s_{n-6} & -s_{n-5} & -s_{n-4} & -s_{n-3} & -s_{n-2} & -s_{n-1} & s_0
\end{array}\right|$$

Before we perform the two row-wise multiplications $\mathbf{h}_1 = \mathbf{cs}_1$ and $\mathbf{h}_2 = \mathbf{cs}_2$ we initialize each coefficient of $\mathbf{h}_1$ and $\mathbf{h}_2$ with 32 to avoid underflow when subtracting (each double is initialized with 141845657554976).

8.2.3 Evaluation and Future Work

As a result, the vectorized and packed implementation of the sparse multiplication (without the improved sampling of $\mathbf{y}_1$ and $\mathbf{y}_2$) allows signing in 572,889 cycles compared to 634,988 cycles required in [GOPS13]. To store the rotations of the secret key $2 \cdot 8{,}288 = 16{,}576$ bytes of memory are required. However, this implementation could not outperform a multiplication in which the secret keys were stored in the NTT domain, as detailed in Section 8.2.1. The cycle counts for our improved implementation (including the improved uniform sampling) and the original values from [GOPS13] are given in Table 8.1. Due to the faster random sampling and storage of keys in the NTT domain, signing is 1.4 times faster in our work compared to [GOPS13]. The verification step is improved by a factor of 1.3.

For comparison other signatures schemes are also listed in Table 8.1 but as they were not all measured on the same platform the results are not all directly comparable. The fastest scheme with regard to signing and also with the smallest signature (5.6 kb) is currently BLISS (implemented in plain C) due to the low amount of rejections, fast Gaussian sampling using a large CDT, and small parameters for n and q. The structural disadvantage of GLP (more rejections, larger n and q) is nearly compensated by the usage of assembly optimization and vectorization (i.e., AVX extensions). As verification almost only requires polynomial multiplication, our GLP implementation is almost three times faster than BLISS. Note also that for the signing procedure of BLISS, the impact of higher security levels on performance is moderate as n and q stay the same, with the significant changes being in the Gaussian sampler and number of rejections. As Gaussian sampling is not needed for verification, the runtime of verification is basically independent of the security level. The implementation of LYU signatures [Lyu12] from [WHCB13] is not competitive, mainly due to larger parameters and also because the implementation uses slow rejection sampling and relies on the number theory library (NTL) for basic arithmetic. For GPV [GPV08], initial outputs and key sizes were many megabits long and even with improvements from [MP12] the signature and key sizes of the implementation by Bansarkhani and Buchmann [BB13] are still large in practice.

Regarding future work, an implementation of the BLISS signature scheme that uses the vectorization ideas from [GOPS13] should lead to a significant performance improvement. In general, the vectorized implementation of the NTT given in [GOPS13] could benefit a whole range of ideal lattice-based schemes. Moreover, it would be worth investigating how efficiently the secure NTRU signature scheme proposed in [MBDG14] can be implemented.

8.3 Implementation of BLISS on the Cortex-M4F

In this section we discuss our implementation of BLISS on the Cortex-M4F and evaluate different methods to sample from a discrete Gaussian distribution. Additionally, we show how the NTT can be used to accelerate the costly BLISS key generation. Note that possible future work on the implementation of BLISS on microcontrollers is discussed in Section 8.4.4.

Our target device is the Cortex-M4F. It is currently the second most powerful device in the Cortex-M series which consists of five 32-bit RISC processors (M0, M0+, M3, M4, M7)[2]. It provides 21 core registers, divided into 13 general purpose registers, 5 special registers, as well as a stack pointer, link register, and program counter register. The core features a floating point

---

[2] See http://www.arm.com/products/processors/cortex-m/

Table 8.1: Comparison of performance and signature size of selected post-quantum signature software implementations on microprocessors.

Software	Platform	Sec.	Cycles		Sizes		
		bits	sign	verify	pk	sk	sig
GLP(**our work**)	Intel Core i5-3210M	80	452,223	34,004	1,536	8,192	1,184
GLP [GOPS13]	Intel Core i5-3210M	80	634,988	45,036	1,536	4,096	1,184
BLISS-I [DDLL13a]	Intel Core i7	128	424,600	102,000	875	250	700
BLISS-III [DDLL13a]	Intel Core i7	160	690,200	105,400	875	375	750
BLISS-IV [DDLL13a]	Intel Core i7	192	1,275,000	108,800	875	375	813
BG [DBG$^+$14]	Intel Core i5-3210M	128	1,973,610	608,870	1,619,940	912,380	1,495
GPV-matrix [BB13]	AMD Opteron 8356	100	287,500,000	48,300,000	24,192,000	11,232,000	27,400
GPV-poly [BB13]	AMD Opteron 8356	100	71,300,000	9,200,000	47,300	23,900	30,100
PASSSign [HPS$^+$14]	Intel Core i7-2640M	130	584,230	172,641	1,500	-	2,360
LYU [WHCB13]	AMD Opteron 8356	80	93,633,000	13,064,000	13,087	13,240	8,294
mqqsig160 [GØJ$^+$11]	Intel Core i5-3210M	80	1,996	33,220	206,112	401	20
mqqsig256 [GØJ$^+$11]	Intel Core i5-3210M	128	4,560	87,904	789,552	593	32
tts6440 [CCC$^+$09]	Intel Core i5-3210M	80	33,728	49,248	57,600	16,608	43
Parallel-CFS [LS12]	Intel Xeon W3670	80	4,200,000,000	-	20,968,300	4,194,300	75
XMSS [BDH11]	Intel i5-M540	82	7,261,100	556,600	912	19	2,451
RSA2048 [DDLL13a]	Intel Core i7	112	4,012,000	129,200	250	250	250
RSA4096 [DDLL13a]	Intel Core i7	128	29,444,000	469,200	500	500	500
ed25519	Intel Core i5-3210M	112	67,564	209,328	32	64	64
ronald2048	Intel Core i5-3210M	112	5,768,360	77,032	256	2,048	256

pk stands for public key; sk stands for private key. The sizes are given in bytes.

unit (FPU) that makes the main difference between Cortex-M4 and Cortex-M4F and allows execution of floating point addition or multiplication in one cycle. Devices from the Cortex-M4 series can also use a digital signal processor (DSP) that is capable of performing instructions like multiplication with subsequent addition in a single cycle. For generation of randomness we rely on the on-board true random number generator (TRNG) which is equipped with a fault detector and employs ring oscillators and a linear feedback shift register (LFSR). The TRNG runs with 48 MHz and can output 32 random bits every 40 periods.

8.3.1 Implementation of Different Discrete Gaussian Samplers

For each signing attempt in BLISS$_{\text{sign}}$ (see step 2 of Algorithm 18) it is necessary to generate $2n = 1024$ discrete Gaussian distributed coefficients with standard deviation $\sigma \approx 215$. As promising candidates for this purpose, we evaluate the Knuth-Yao [RVV13, DG14] and discrete Ziggurat [BCG$^+$13] sampling algorithms as well as an approach using Bernoulli distributed variables [DDLL13a] (see Section 2.5.3).

The actual implementation of the Bernoulli sampler is straightforward. The algorithm requires to store 336 bytes of precomputed data and we use loop unrolling to speed up certain parts of the algorithm. This reduces the average required cycles from 2,049 to 1,835 cycles per Gaussian distributed coefficient. Our implementation of the Knuth-Yao algorithm is inspired by [RVV13] which has been adapted to software. Necessary tables were computed using the computer algebra system SAGE and we removed a large number of leading zeros that occur in the first columns. The overall memory consumption is 19,064 bytes which is still large but favorable compared to a naive CDT approach, which would require 41,280 bytes. However, the detection of leading zeros leads to some overhead so that we need on average 2,404 cycles per sample. The main challenge of an implementation of the Ziggurat algorithm is its computational complexity and the infrequent high precision rejection sampling. Thus we had to implement expensive multi-precision arithmetic and compute the exponential function using limit representation. We refer to [DG14] for a short survey on methods to compute exp().

8.3.2 Polynomial Arithmetic

Besides discrete Gaussian sampling, polynomial arithmetic is one of the most time-consuming parts of BLISS. In this section we concentrate on the costly multiplication and inversion in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$.

Number Theoretic Transform

For polynomial multiplication we use the NTT and refer to Section 2.4.2 for the theoretical background. We precompute all necessary twiddle factors required by the NTT and have unrolled the first two stages of the implemented radix-2 decimation-in-time algorithm (DIT) (see Algorithm 1). The core of the algorithm is implemented in assembly and listed in Table 8.3. We also use conditional execution and powerful DSP instructions, like multiply-subtract, to speeds up our implementation.

Another useful feature of our target microcontroller is a bit-reversal instruction (`RBIT`). This is necessary as the input to the transform has to be reordered such that coefficients are exchanged with their bit-reversed counterpart (see Section 2.4.2). With the help of the `RBIT` instruction, we can implement this step efficiently using inline assembly. Another option would have been the usage of the optimizations discussed in Section 6.2 to render bit-reversal unnecessary altogether.

Polynomial Inversion

During the key generation algorithm $\mathsf{BLISS}_{\mathsf{gen}}$, it is necessary to compute the multiplicative inverse of $\mathbf{f}$ (Algorithm 17, step 5). For this task we use Fermat's little theorem to compute the multiplicative inverse as $\mathbf{f}^{-1}=\mathbf{f}^{q-2}$ in $\mathcal{R}_q$ and an addition chain [Knu97] that requires 18 polynomial multiplications[3]. Processing the exponent $12289-2=12287=10111111111111_2$ bitwise via the square-and-multiply algorithm [VOMV96] would need 25 polynomial multiplications what makes the addition chain approach preferable. All operations during the inversion are performed in the frequency domain and it suffices to compute the NTT transformation at the beginning and the end of the exponentiation. This provides the possibility to apply an early

---

[3] See ACHAIN-ALL `http://www-cs-faculty.stanford.edu/~uno/programs.html`. We would like to thank Léo Ducas for helpful discussions on this topic and for pointing us to the ACHAIN-ALL program.

Table 8.3: Implementation of the NTT butterfly operation of Algorithm 1 in C (on the left) and assembly (on the right) on the ARM Cortex-M4F.

C Code	Assembler
// omega[m] in LR	ldr.W LR, [R6]
// out[b] in R9	ldr.W R9, [R0,R12,LSL #2]
r =(omega[m]*out[b])%q;	mul LR, R9, LR
// q in R11 and R8	mov R11, R8
// out[a] in R9	ldr.W R9, [R0,R7,LSL #2]
// out[b] in R10	ldr.W R10, [R0,R12,LSL #2]
	sdiv R11, LR, R11
// result mod q in R11	mls R11, R8, LR
out[b]=out[a]+(q-r)%q;	subS.W R10, R9, R11
	IT MI
	addMI R10, R10, R8
out[a]=out[a]+r % q;	add R9, R9, R11
	cmp R9, R8
	IT GE
	subGE.W R9, R9, R8
// write back out[b]	str.W R10, [R0,R12,LSL #2]
// write back out[a]	str.W R9, [R0,R7,LSL #2]

test for invertibility of the input, because we can simply check whether there are coefficients that are equal to 0 after transforming the input into the frequency domain. Another advantage of the frequency domain is that polynomial multiplication is just coefficient-wise multiplication. We can exploit this to minimize the memory consumption by computing the addition chain iteratively for all coefficients. Thus we do not have to store whole polynomials as intermediate results but just one coefficient.

Sparse Multiplication

In step 6 and step 7 of Algorithm 18, the computation of $\mathbf{s}_1\mathbf{c}$ and $\mathbf{s}_2\mathbf{c}$ is required. Since $\mathbf{c}$ is only a sparse polynomial where κ coefficients are set to one, applying the NTT is not necessarily the optimal solution. Moreover, we do not need to reduce modulo $2q$ as the polynomials $\mathbf{s}_1, \mathbf{s}_2$ have only small coefficients. For efficiency reasons we therefore only store the index of the coefficients of $\mathbf{c}$ that are one and need roughly κn additions in $\mathbb{Z}_q$. We further decreased the runtime of the sparse schoolbook multiplication from 354,419 to 224,626 cycles by unrolling the inner loop at the cost of a code size increased by 602 bytes.

8.3.3 Results

In this section, we present performance results for our implementation of BLISS-I on the Cortex-M4F. The Cortex-M4F microcontroller operates at 168 MHz and our code is compiled using IAR Embedded Workbench for ARM in version 6.60.1.5104. For precise benchmarks, we determined average cycle counts of a subroutine with random inputs from 1,000 runs and used a data watchpoint trigger to exactly evaluate the clock cycle counter.

Performance Results

Cycle counts for major building blocks of BLISS are given in Table 8.4. The results show that computations required during key generation are expensive in terms of runtime and RAM consumption. However, the polynomial inversion is even 1.08 times faster than a single NTT-based polynomial multiplication since we preserve the NTT-transformed representation between different functions and use the addition chain to speed up the inversion. By using the sparse multiplication, we only need 54.9% of the cycles of an NTT multiplication on the same values. Referring to the results of Section 8.2, this shows that the advantage of using sparse multiplication over the NTT highly depends on the target architecture. We evaluated three instantiations of the Ziggurat algorithm, one for a rather small precomputed table with a size of 2,560 bytes, one as a time-memory trade-off with a precomputed table of 5,120 bytes, and one for a rather large precomputed table with a size of 10,240 bytes for time-critical applications. The trade-off sampler is 1.95 times faster than the size-optimized sampler whereas the speed-optimized sampler is 3.19 times faster than the trade-off sampler. But even the speed-optimized variant of the Ziggurat sampler is not able to outperform the Bernoulli sampler. The generation of the signature component **c**, which also includes the KECCAK hash function [BDPA13], performs well compared to other subroutines. The removal of lower bits of $\mathbf{z}_2$ is negligible with about 8,000 cycles.

Table 8.4: Performance measurement of the major building blocks of our BLISS-I implementation on the Cortex-M4F.

Routine	Cycles	Application
NTT Trans.	122,619	g/s/v
NTT Multiplication	508,624	g/s/v
Polynomial Inversion	470,606	g
Computation of $\mathrm{N}_\kappa(\mathbf{S})$	1,043,447	g
Generate **c**	220,022	s/v
Drop bits	8,225	s/v
Sparse Multiplication	224,626	s
Huffman Encoding	78,927	s
Huffman Decoding	115,943	v
Sampling Bernoulli	935,925	s
Sampling Knuth-Yao	1,231,326	s
Sampling $\text{Ziggurat}_{\text{speed}}$	1,057,814	s
Sampling $\text{Ziggurat}_{\text{average}}$	1,729,098	s
Sampling $\text{Ziggurat}_{\text{size}}$	3,378,909	s

Note that the NTT transformation is applied on polynomials with $n = 512$ coefficients. Gaussian distributed values are sampled from $D_{\mathbb{Z}^n,\sigma}$ for $n = 512$ and $\sigma = 215$. We denote by g/s/v if a routine is used in key (g)eneration, (s)igning, or (v)erification.

The number of clock cycles and memory consumption for key generation, signing and verification are given in Table 8.6. Key generation is a rather slow process since it needs to be restarted frequently and requires time-consuming computations (see Table 8.4). The performance of the signing operation is directly determined by the chosen sampler. We observe that the RAM consumption of the signing operation is independent from the sampling algorithm since all samplers need a comparatively small amount of memory. The table used for the Bernoulli sampler is also used for the computation of the hyperbolic cosine function. The total amount of flash memory includes 900 bytes for the Bernoulli sampler. Additional flash memory is also consumed by code that initializes peripherals of the Cortex-M4F and is responsible for debug and profiling output.

All in all our results show that the Bernoulli sampler is clearly the best choice compared to the other evaluated samplers. It provides the lowest runtime and needs the smallest precomputed table compared to the Ziggurat and Knuth-Yao samplers. It would certainly be possible to speed up the Ziggurat with even larger tables but for most constrained devices this is not an option. A major issue with the Ziggurat sampler is the requirement for multi-precision arithmetic. Our implementation of the Knuth-Yao sampler is the least favorable choice with respect to performance and memory consumption compared to the Bernoulli and the speed-optimized Ziggurat sampler. A major drawback of this algorithm is the large table and the high amount of memory accesses that slow down the implementation. These results are also counter-intuitive, since we expected the algorithm with the *largest* table to outperform the others. But instead, the Bernoulli sampler as the one with the *smallest* table is the preferable solution according to our results.

Table 8.6: Results for our implementation of key generation, signing, and verification of BLISS-I on the Cortex-M4F.

Operation	Cycles	RAM	Flash
$\text{Signing}^{\text{Ber}}$	5,927,441	18,580	24,648
$\text{Signing}^{\text{KY}}$	6,865,089	18,580	44,036
$\text{Signing}^{\text{Zig-Speed}}$	5,984,686	18,580	36,028
$\text{Signing}^{\text{Zig-Average}}$	8,335,711	18,580	30,908
$\text{Signing}^{\text{Zig-Size}}$	16,396,414	18,580	28,420
Verification	1,002,299	11,520	-
Key Generation	367,859,092	21,272	-

Note that the flash consumption for the signing algorithms already includes the code and tables for verification and key generation.

Performance Comparison

In Table 8.8 we provide results obtained from the documentation of the STM32 Cryptographic Library [STM] which is evaluated on a STM32F4xx family (Cortex-M4) microcontroller. There are also other results for ARM-based microcontrollers in the literature, but many implementations run on outdated hardware (e.g., [AYK00] on ARM7TDMI) and do not allow a fair comparison. A recent work evaluates ECC on a much less powerful Cortex-M0, with a special focus on energy consumption [dCUHV13]. A comparison with the CPU implementation of BLISS given in [DDLL13a] is certainly not fair as well. Besides architectural differences the biggest advantage of desktop CPUs is that much more memory is available so that Gaussian sampling can be implemented using algorithms that do not optimize for storage efficiency.

In terms of security, the implemented signature scheme was designed to provide a level of 128 bits of equivalent symmetric security. It can be compared to RSA-2048 (or maybe even RSA-4096) and ECC-256. In terms of speed our verification routine outperforms RSA and ECC for all common security parameters. Moreover, our implementation is twice as fast compared to ECC-256 regarding signature generation. It becomes also obvious that RSA gets impractical for parameter sets larger than RSA-2048 on constrained devices, especially due to the very slow signing. With regard to signature size we achieve the 5,600 bits stated in [DDLL13a] by using the same approach to Huffman encoding as described in Section 7.3.3 for a hardware implementation. The public key requires 1,024 bytes and the private key requires 384 bytes. For n-RSA signatures the size of the signature is $\frac{n}{8}$ bytes and for n-ECC signatures (ECDSA) the signature size is $2\frac{n}{8}$ bytes.

Table 8.8: Comparison of the most efficient instantiation of our implementation with the RSA and ECC implementation of the STM32 Cryptographic Library (target device: STM32F4xx family) [STM].

	Cycles		
Operation	Key generation	Sign	Verify
BLISS$^{\text{Ber}}$	367,859,092	5,927,441	1,002,299
RSA-1024	-	30,627,432	1,573,079
RSA-2048	-	228,068,226	6,195,481
ECC-192	7,400,421	7,720,020	14,716,374
ECC-224	9,849,334	10,414,487	19,558,528
ECC-256	12,713,277	13,102,239	24,702,099

8.4 Implementation of BLISS on the ATxmega

In this section we discuss an implementation of BLISS on the ATxmega platform. Our results show superior performance compared to related work and that BLISS is practical even on 8-bit architectures.

8.4.1 Implementation of BLISS Using the NTT

For our implementation we do not introduce new techniques but rely on the most promising CDT-based discrete Gaussian sampler (see Section 2.5.4 and Section 7.3.1) using convolution and Kullback-Leibler divergence as well as the optimized NTT algorithms $\text{NTT}^{CT,\psi}_{no\rightarrow bo}$ and $\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ (see Section 6.2) used to implement RLWEenc.

Table 8.9: Cycle counts and flash memory consumption in bytes for the implementation of BLISS on an 8-bit ATxmega128 microcontroller.

Operation	(n=512, q=12289)
	Cycle counts and stack usage
$\mathsf{BLISS}_{\mathsf{sign}}$	10,537,981 (4,012 bytes)
$\mathsf{BLISS}_{\mathsf{verify}}$	2,814,118 (1,103 bytes)
$\mathrm{NTT}_{no\rightarrow bo}^{CT,\psi}$	521,872
$\mathrm{INTT}_{bo\rightarrow no}^{GS,\psi^{-1}}$	497,815
SampleGauss	1,140,600
SparseMul	503,627
Hash	1,335,040
GenerateC	4,410
DropBits	11,826
$\cosh\left(\frac{\langle \mathbf{z},\mathbf{Sc}\rangle}{\sigma^2}\right)$	75,601
$M\exp\left(-\frac{\lVert\mathbf{Sc}\rVert^2}{2\sigma^2}\right)$	37,389
	Static memory consumption in bytes
Complete binary	18,674
RAM	2,411

The stack usage is divided into a fixed amount of memory necessary for message, signature, and additional components (like random number generation) and the dynamic consumption of the signing and verification routine. We sign a message of n bits.

To realize the reduction mod $2q$, we create a second reduction function by simply extending the approach from Figure 6.3. The difference is that for the modulus $2q$, `v` is loaded with the initial value `12289 « 16` and therefore two additional steps `REDUCE30` and `REDUCE29` have to be inserted. Finally, we exclude the last step that subtracts q to get a result in $[0, 2q]$. For the instantiation of the random oracle (Hash) that is required during signing and verification we have chosen the official AVR implementation of KECCAK [BDP+12]. From the output of the hash function the sparse polynomial $\mathbf{c}$ with κ coefficients equal to one is generated by the GenerateC (see [DDLL13b, Section 4.4]) routine. We store only κ indices where a coefficient of $\mathbf{c}$ is one. This reduces the dynamic RAM consumption and allows a more efficient implementation of the multiplication of $\mathbf{c}$ by $\mathbf{s}_1$ and $\mathbf{s}_2$ using the SparseMul routine. By using column-wise multiplication and by ignoring all zero coefficients, the multiplication can be performed more efficiently than with the NTT.

8.4.2 Results

In Table 8.9, we present detailed cycle counts for signing and verifying as well as for the most expensive operations in BLISS-I. Due to the rejection sampling and the chosen parameter set, 1.6 signing attempts are required on average to create one signature. One attempt requires 6,381,428 cycles on average and only a small portion of the computation, i.e. the hashing of the message, does not have to be repeated in case of a rejection. During a signing attempt

Table 8.11: Comparison of our AVR implementation of BLISS with related work.

Scheme	Device	Operation	Cycles		OP/s	
BLISS-I, **our work**	AX128	Sign/Verify	10,537,981	2,814,118	3.04	11.37
BLISS-I (Bernoulli) [BSJ14]	AT64	Sign+Verify	42,069,682*		0.19	
BLISS-I (CDT) [BJ14]	AX64	Sign+Verify	19,328,000*		1.65	
Ed25519† [HS13]	AT2560	Sign/Verify	23,211,611	32,619,197	0.67	0.49
RSA-1024 [GPW$^+$04]	AT128	Sign/Verify	3,440,000	87,920,000	2.33	0.09
RSA-1024† [LGK10]	AT128	priv. key	75,680,000		0.11	
ECC-ecp160r1 [GPW$^+$04]	AT128	Point mul.	6,480,000		1.23	

Cycle counts marked with (*) indicate the runtime of BLISS-I as authentication protocol instead of signature scheme. By AX128 we identify the ATxmega128A1 clocked with 32 MHz, by AX64 the ATxmega64A3 clocked with 32 MHz, by AT64 the ATmega64 clocked with 8 MHz, by AT128 the ATmega128 clocked with 8 MHz, and by AT2560 the ATmega2560 clocked with 16 MHz.

the most expensive operation is the sampling of two Gaussian distributed polynomials which takes $2 \times 1{,}140{,}600 = 2{,}281{,}200$ cycles (36% of the overall cycles). The calls to $\mathrm{NTT}_{no\rightarrow bo}^{CT,\psi}$ and $\mathrm{INTT}_{bo\rightarrow no}^{GS,\psi^{-1}}$ account for 16% of the overall cycles of one attempt. In contrast to the RLWEenc implementation we do not use a look-up table for the first stage of $\mathrm{NTT}_{no\rightarrow bo}^{CT,\psi}$. Additionally, we do not implement a separate modulo reduction after the subtraction in the GS butterfly and reduce the result after the multiplication which explains the slightly better performance of $\mathrm{INTT}_{bo\rightarrow no}^{GS,\psi^{-1}}$. Hashing the compressed $\mathbf{u}$ and the message μ is time consuming and accounts for roughly 21% of the overall cycles during one attempt. Savings would be definitely possible by using a different hash function (see [BEE$^+$12a] for an evaluation of different functions) but KECCAK appears to be a conservative choice that matches the 128-bit security target very well. The sparse multiplication takes only 503,627 cycles for one multiplication. This makes it a favorable approach on the ATxmega and overall the sparse multiplication is 3.6 times faster than an NTT multiplication approach that would require one $\mathrm{NTT}_{no\rightarrow bo}^{CT,\psi}$, two $\mathrm{INTT}_{bo\rightarrow no}^{GS,\psi^{-1}}$ and two PwMulFlash calls. The flash memory consumption includes $2n$ words which equals $4n = 2{,}048$ bytes for the NTT twiddle factors and 3,374 bytes for look-up tables of the sampler.

8.4.3 Comparison

Our results for BLISS and results of related works are provided in Table 8.11. A comparison between our implementation of BLISS-I and the implementations of [BSJ14] and [BJ14] is difficult since the authors implemented the signature as authentication protocol. Therefore, they only provide the runtime of a complete protocol run that corresponds to one signing operation and one verification in our results, but without the expensive hashing as the sparse polynomial $\mathbf{c}$ is not obtained from a random oracle but randomly generated by the other protocol party. However, our implementations of $\mathsf{BLISS}_{\mathsf{sign}}$ and $\mathsf{BLISS}_{\mathsf{verify}}$ still require less cycles than their implementation. The biggest improvement stems from the usage of the CDT using convolutions and Kullback-Leibler divergence (see Section 2.5.4) which is superior (1,140,600 cycles per polynomial) compared to the Bernoulli approach (13,151,929 cycles per polynomial [BSJ14]) and the straight-forward CDT approach used in [BJ14]. As our implementation of BLISS-I needs 18,802 bytes of flash memory, it is also smaller than the implementation of [BJ14] that requires 66.5 kB of flash memory and the implementation of [BSJ14] that needs 25.1 kB of flash memory. Our 128-bit secure BLISS-I implementation is 2.2 times faster for signing and 11.6 faster for verification compared to an implementation of the Ed25519 signature scheme for an ATmega2560 [HS13].

8.4.4 Conclusion and Future Work

In this work we have shown that it is possible to implement a post-quantum lattice-based signature scheme on a Cortex-M4F and ATxmega microcontroller with a reasonable flash and memory consumption as well as on an.

Necessary future work to facilitate practical adoption of BLISS and other lattice-based DSSs would be the evaluation of the resistance against side-channel attacks and the implementation of side-channel and fault analysis countermeasures on constrained devices. Additionally, the implementation of the NTT can be further improved as shown in [dCRVV15] where the authors present an implementation of RLWEenc public-key encryption. We also expect that an application of the NTT techniques discussed in Section 6.2 and improvements to Gaussian sampling discussed in Section 8.4 would lead to increased performance.

Chapter 9

Acceleration of Homomorphic Evaluation on Reconfigurable Hardware

Homomorphic encryption allows computation on encrypted data and makes it possible to securely outsource computational tasks to untrusted environments. However, the necessary computations are complex and time consuming even on state of the art microprocessors. To reduce the runtime of homomorphic evaluation operations we propose a hardware accelerator for the YASHE *somewhat homomorphic encryption (SHE) scheme. For efficient utilization of the external memory we describe a double-buffered memory access scheme and a polynomial multiplier based on the number theoretic transform (NTT). This chapter is based on [PNPM15a, PNPM15b] and the work was carried out while the author of this thesis was an intern in the Cryptography Group at Microsoft Research Redmond.*

Contents of this Chapter

9.1 Introduction

A homomorphic encryption scheme enables a third party to perform meaningful computation on encrypted data and a prime example for an application is the outsourcing of a computational task into an untrusted cloud environment (see, e.g., [NLV11, BLN14, CKK15, CKL15]). Such schemes come in different flavors, the most versatile being a fully homomorphic encryption (FHE) scheme, which allows an unlimited number of operations. The first FHE scheme was proposed by Gentry in 2009 [Gen09] and led to a large number of new schemes optimized for better efficiency or security (e.g., [vDGHV10, CMNT11, GH11, BGV12, Bra12, LTV12, GSW13]).

FHE schemes usually consist of a so-called somewhat or leveled homomorphic scheme with limited functionality together with a procedure to bootstrap its capabilities to an arbitrary number of operations. The SHE schemes are usually a lot more efficient than their corresponding FHE counterparts because bootstrapping imposes a significant overhead. Examples of SHE schemes are the Brakerski-Gentry-Vaikuntanathan (BGV) [BGV12] and López-Alt-Tromer-Vaikuntanathan (LTV) [LTV12] schemes and the subsequent YASHE (yet another somewhat homomorphic encryption) [BLLN13b] scheme which are relatively straightforward and conceptually simple as they mainly require polynomial multiplication and (bit-level) manipulation of polynomial coefficients for evaluation of ciphertexts (i.e., multiplication and addition). But even limited SHE schemes are still slow and especially for relatively complex computations, evaluation operations can take several hours, even on high-end CPUs [GHS12, LN14]. A natural question concerning FHE and SHE is whether reconfigurable hardware can be used to accelerate the computation. However, as ciphertexts and keys are large and require several megabytes or even gigabytes of storage for meaningful parameter sets, the internal memory of FPGAs is quickly exhausted, and required data transfers between host and FPGA might degrade the achievable performance.

These may be reasons that previous work has mainly focused on using graphics cards [WCH14, DÖS15, WHC$^+$15] and application specific integrated circuits (ASIC) [DÖS13, WHEW14], and that FPGA implementations either work only with small parameters and on-chip memory (see [CMV$^+$14]) or explicitly do not take into account the complexity of transferring data between an FPGA and a host (see [CMO$^+$14, RJV$^+$15]). Also, for ASIC implementations, the area costs for caches and on-chip storage are high due to large parameter sizes. As an example, in [DÖS13] 99.25 % of the 26.7 million gates are used to implement a 768 Kbyte cache. As a consequence, for our implementation we use the Catapult data center acceleration platform [PCC$^+$14], which provides a Stratix V FPGA on a board with two 4 GB memory modules inserted into the expansion slot of a cloud server. This fits nicely into the obvious scenario in which homomorphic evaluation operations are carried out on encrypted data stored in the cloud. Since future data centers might be equipped with such accelerators, it makes sense to consider the Catapult architecture as a natural platform for evaluating functions with homomorphic encryption.

Contribution. We provide a fully functional FPGA implementation of the homomorphic evaluation operations of a RLWE based SHE scheme. Our main contribution is an efficient architecture for performing number theoretic transforms, which is used to implement the SHE scheme YASHE (see Section 3.7). Compared to previous FPGA implementations of integer-based FHE schemes (e.g., [CMO$^+$14]), we especially take into account the complexity of using off-chip memory. Thus we propose and evaluate the usage of the cached-NTT [Baa99, Baa05] for bandwidth-efficient computations of products of large polynomials in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$ and the YASHE specific parts of the KeySwitch and Mult algorithms. The main computational burden is handled by a large integer multiplier built out of DSP blocks and modular reduction using Solinas primes. An implementation of the parameter set ($n = 16384, \lceil \log_2 q \rceil = 512$) that can handle computations on the encrypted data of multiplicative depth up to $L = 9$ levels (for $t = 1024$) roughly matches the performance of a software implementation of the parameter set ($n = 4096, \lceil \log_2 q \rceil = 128$) supporting just one level [LN14]. With only 48.67 ms for a homomorphic multiplication (instead of several seconds in software), we provide evidence that hardware-accelerated SHE can be made practical for certain application scenarios.

9.2 Background

In this section we introduce the cached-NTT and the Catapult framework. Note that YASHE is described in Section 3.7.

9.2.1 Cached-FFT

The cached-FFT algorithm has been proposed by Baas [Baa99, Baa05]. It is designed for systems with hierarchical memory like modern processors that usually have several layers of fast caches (e.g., L1, L2) and then relatively slow main memory, e.g., dynamic random access memory (DRAM). It is assumed that the cache that contains C coefficients can be accessed faster and with much less latency than the rest of the memory and that the cache is much smaller than the total number of coefficients n. The high-level description of the steps of the cached-FFT or cached-NTT (obtained from [Baa05], with small updates and adapted notation) is:

(1) n input coefficients are loaded into main memory.

(2) C of the n coefficients are loaded into the cache.

(3) As many butterflies as possible are computed using the data in the cache.

(4) Processed data in the cache is flushed to main memory.

(5) Steps 2–4 are repeated until all n words have been processed once.

(6) Steps 2–5 are repeated until the FFT has been completed.

The following definitions (also obtained from [Baa05] with small updates and adapted notation) are useful when describing the cached-FFT algorithm:

- An epoch (E) is the portion of the cached-FFT algorithm where all n coefficients are loaded into the cache, processed, and written back to main memory once. Normally, $E \geq 2$. Steps 2–5 in the listing above comprise the computations performed on one epoch.
- A group (G) is the portion of an epoch where a block of data is read from main memory into the cache, processed, and written back to main memory. Steps 2–4 in the listing above comprise the operations performed on a group. A group contains C coefficients.
- A pass (P) is the portion of a group where each word in the cache is read, processed with a butterfly, and written back to the cache once.
- A cached-FFT is balanced if there are an equal number of passes in the groups from all epochs. Balanced cached-FFTs do not exist for all FFT lengths.

When describing an FFT or NTT we thus denote the number of epochs by E, the number of groups by G, the number of coefficients in the cache by C, and the number of passes by P. The required computation on a group is just a standard Cooley-Tukey, radix-2, in-place, decimation-in-time FFT/NTT [CT65, CG00], denoted as CT-NTT and the number of stages or passes (recursive divisions into sub-problems) of the CT-NTT is $P = \log_2(n/G)$. Thus one CT-NTT on a group requires $\frac{Pn}{2G}$ multiplications in $\mathbb{Z}_q$. A dataflow diagram of a 64-point radix-2

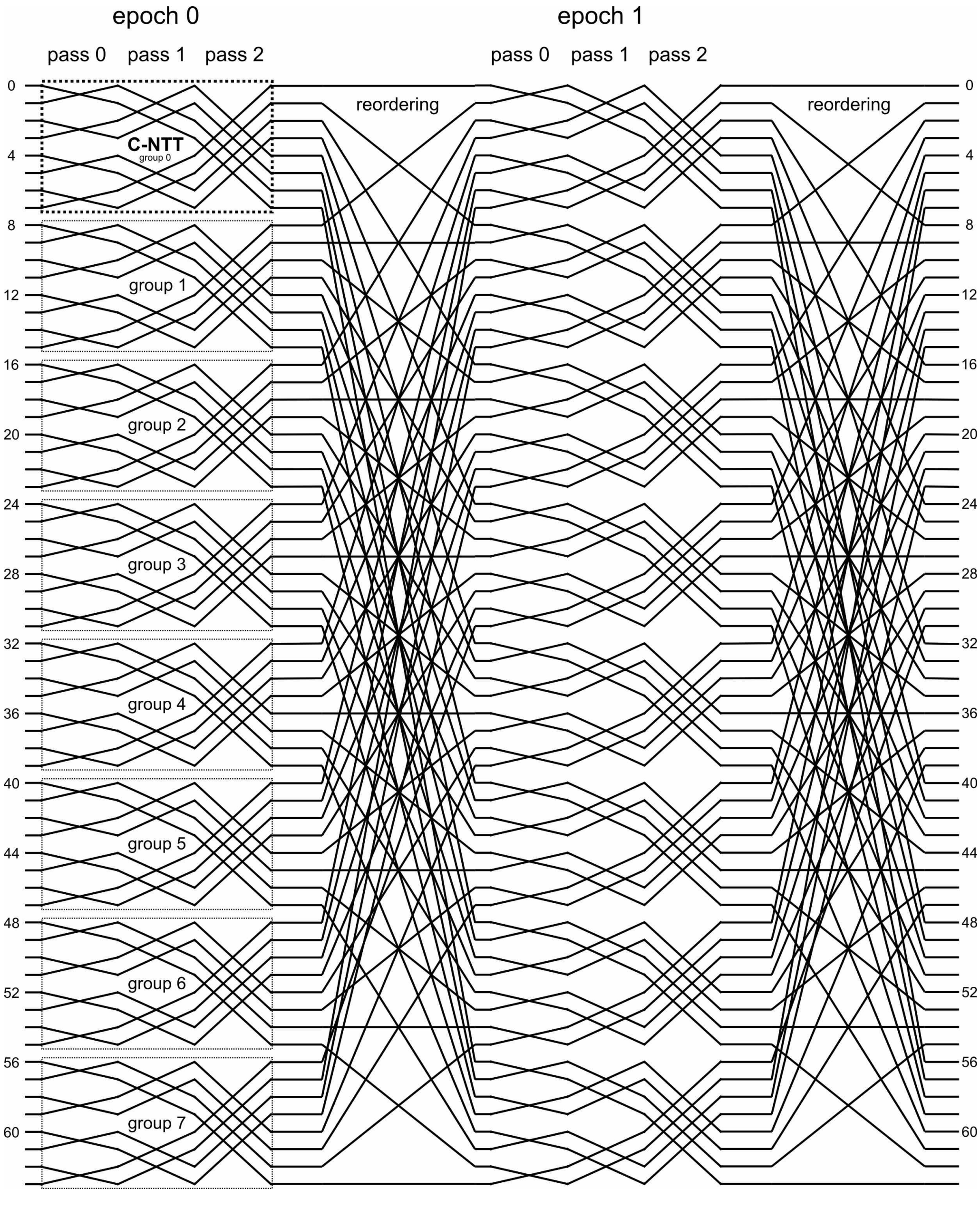

Figure 9.1: Dataflow diagram, based on [Baa05, Figure 3], of a 64-point cached-FFT split into two epochs with eight coefficients in each group/cache parameterized as $(n{=}64, E{=}2, C{=}8, G{=}8, P{=}3)$.

Table 9.1: Configuration options (n, E, C, G, P) of the cached-FFT for various values of n usually used in RLWE-based homomorphic cryptography.

Dimension (n)	Epochs (E)	Cache size ($C = n/G$)	Groups ($G = n/C$)	Passes per epoch ($P = \log_2(n)/E$)
2048	1	2048	1	11
2048	11	2	1024	1
4096	1	4096	1	12
4096	**2**	**64**	**64**	**6**
4096	3	16	256	4
4096	4	8	512	3
4096	6	4	1024	2
4096	12	2	2048	1
8192	1	8192	1	13
8192	13	2	4096	1
16384	1	16384	1	14
16384	**2**	**128**	**128**	**7**
16384	7	4	4096	2
16384	14	2	8192	1
32768	1	32768	1	15
32768	3	32	1024	5
32768	5	8	4096	3
32768	15	2	16384	1
65536	1	65536	1	16
65536	2	256	256	8
65536	4	16	4096	4
65536	8	4	16384	2
65536	16	2	32768	1

DIT cached-FFT that splits the computation into two epochs, each consisting of eight groups, is given in Figure 9.1. Different configuration options of the cached-FFT for dimensions $n \geq 2048$ are given in Table 9.1 where C, G, P depend on the chosen number of epochs E for a given dimension n. For the actual details of the implementation of address generation we refer to the description in [Baa99, Baa05]. However, referring to the $E = 2$ case displayed in Figure 9.1, it is easy to see that, with a hardware implementation in mind, it is necessary to read $2n$ coefficients from the main memory and to write $2n$ coefficients back to the main memory to compute the FFT. However, only two of these reads/writes are non-consecutive (i.e., the reordering) while two read/writes are in order. Additionally, it is possible, e.g., after epoch zero, to choose whether to write consecutive and then to read reordered or the other way round. Another interesting observation for the $E = 2$ case is that the twiddle factors in epoch zero are the same for each group. When the cached-FFT algorithm is used to implement the NTT algorithm (with built-in reduction by $\mathbf{x}^n + 1$) it is also possible to merge the multiplication by powers of ψ and the actual NTT computation as proposed in [RVM+14, Section 3.2].

A bit-reversal is necessary as the cached-FFT is not order preserving and expects bit-reversed input and produces an ordered output. The functions to compute the bitreversal (BitrevAddr) of an address is given in Algorithm 33 and the computation of the cached-NTT reordering of an address (Reorder) is given in Algorithm 34. Note that both functions assume n to be a power of two.

Algorithm 33 Bit-Reversal Operation

```
func BitrevAddr($x \in [0, 2^n)$)
    //$x$ is an integer in binary form $x = x_{n-1} \cdots x_0$
    for $i = 0$ to $n - 1$ do
        $y_i \leftarrow x_{n-1-i}$
    end for
    return $y$
end func
```

Algorithm 34 Cached-NTT Reordering

```
func Reorder($x \in [0, 2^n)$)
    //$x$ is an integer in binary form $x = x_{n-1} \cdots x_0$
    $y_{n/2-1,\ldots,0} \leftarrow x_{n-1,\ldots,n/2}$
    $y_{n-1,\ldots,n/2} \leftarrow x_{n/2-1,\ldots,0}$
    return $y$
end func
```

9.2.2 Catapult Architecture/Target Hardware

Because a primary application of homomorphic encryption is use in untrusted clouds, we chose to implement YASHE using a previously proposed FPGA-based datacenter accelerator infrastructure called Catapult [PCC$^+$14]. Catapult augments a conventional server with an FPGA card attached via PCI express (PCIe) that features a medium size Stratix V GS D5 (5GSMD5) FPGA, two 4 GB DDR3-1333 SO-DIMM (small outline dual inline memory module) memory modules, and a private inter-FPGA 2-D torus network. In the original work, Catapult was used to accelerate parts of the Bing search engine, and a prototype consisting of 1,632 servers was deployed. The two DRAM controllers on the board can be used either independently or combined in a unified interface. When used independently the DIMM modules are clocked with 667 MHz. The Catapult shell [PCC$^+$14, Section 3.2.] provides a simple interface to access the DRAM and to communicate with the host server. It uses roughly 23% of the available device resources, depending on the used functionality like DRAM, PCIe, or 2-D torus network. Application logic is implemented as a role. For our design, we restrict the accelerator to only a single FPGA card per server. Spanning multiple FPGAs is a promising avenue for improving performance, but is left for future work. Note also that none of the work presented here is exclusive to Catapult and that any FPGA board with two DRAM channels, a sufficiently large FPGA, and fast connection to a host server will suffice. However, Catapult is specifically designed for datacenter workloads, so it presents realistic constraints on cost, area, and power for our accelerator.

9.3 High-Level Description

The goal of our implementation is to accelerate the (cloud) server-based evaluation operations Mult and Add of YASHE (and polynomial multiplication in general) without interaction with the host server using the Catapult infrastructure. Key generation, encryption, and decryption are assumed to be performed on a client and are not in the scope of this work. However, we would like to note that except for a Gaussian sampler, most components required for key generation, encryption, and decryption are already present in our design.

Our main building block is a scalable NTT-based polynomial multiplier that supports the two moduli q and q'. The computation of the NTT is by far the most expensive operation and necessary for the polynomial multiplications in RMult and KeySwitch, which are called during a Mult operation. Other computations like polynomial addition or pointwise multiplication are realized using the hardware building blocks from the NTT multiplier. The modulus $q' > nq^2$ is used to compute

$$\mathbf{c}_1\mathbf{c}_2 = \mathrm{INTT}_{q'}(\mathrm{NTT}_{q'}(\mathbf{c}_1) \circ \mathrm{NTT}_{q'}(\mathbf{c}_2))$$

in RMult exactly without modular reduction as each coefficient of $\mathbf{c}_1$ and $\mathbf{c}_2$ is smaller than q and thus each coefficient of the result is guaranteed to be smaller than nq^2. Reductions modulo q are required for the computation of the scalar product $\mathbf{c}_{\mathrm{mult}} = [\langle \mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\mathrm{mult}}), \mathsf{evk}\rangle]_q$ in KeySwitch and the polynomial addition in Add. A naive implementation of KeySwitch would require $\ell_{w,q}$ polynomial multiplications and $\ell_{w,q}-1$ polynomial additions. By using the NTT and its linearity we just compute

$$\mathsf{KeySwitch}(\tilde{\mathbf{c}}_{\mathrm{mult}}, \overline{\mathsf{evk}}) = \mathrm{INTT}_q\left(\sum_{i=0}^{\ell_{w,q}-1} \mathrm{NTT}_q\left([(\mathbf{c}_{\mathrm{mult}})_i]_w\right) \circ \overline{\mathsf{evk}}_i\right) \tag{9.1}$$

and store the evaluation keys evk_i in NTT form as $\overline{\mathsf{evk}}_i = \mathrm{NTT}_q(\mathsf{evk}_i)$ for $i \in [0, \ell_{w,q}-1]$ (similar to [DÖS15, Algorithm 2]). To deal with the limited internal memory when computing the NTT we use the aforementioned cached-FFT algorithm [Baa99, Baa05]. This enables us to exploit the memory hierarchy on Catapult where we have access to fast but small FPGA-internal memory (≈ 4.9 MiB) and large but slow external DRAM (two times 4 GB). We also incorporate some of the optimizations to the NTT proposed in [RVM+14]. By merging the multiplication by powers of ψ into the twiddle factors of the main NTT computation we not only save n multiplications but also eliminate expensive read and write operations. To optimally utilize the burst read/write capabilities of the DRAM[1] we have designed our core in a way that we balance non-continuous reorderings and continuous reads or writes. While we only implemented two main parameter sets, our approach is scalable and could be extended to even larger parameter sets[2] and is also generally applicable as we basically implement polynomial multiplication, which is common in most RLWE-based homomorphic encryption schemes.

The general architecture of our `HomomorphicCore` design is shown in Figure 9.2. We have divided our implementation into a memory management unit (`NTTMemMgr`) and an NTT computation unit (`NttCore`). The `NTTMemMgr` component performs the steps 2 and 4 of the listing in Section 9.2.1 (load/store of groups) while `NttCore` is responsible for step 3 (butterfly computations on the cache). Both components have access to the memories `ConstDualBuf` and

---

[1] The throughput of the DRAM is drastically increased if large continuous areas of the memory are read at once using the so called *burst mode*.

[2] However, in this case the current tools for RTL-level simulation are clearly a limiting factor as verification in the simulator becomes extremely time consuming. As a consequence, it appears that a generic architecture supporting small parameter sets is almost mandatory to allow relatively efficient simulation and debugging and thus less debug cycles when working on larger parameter sets.

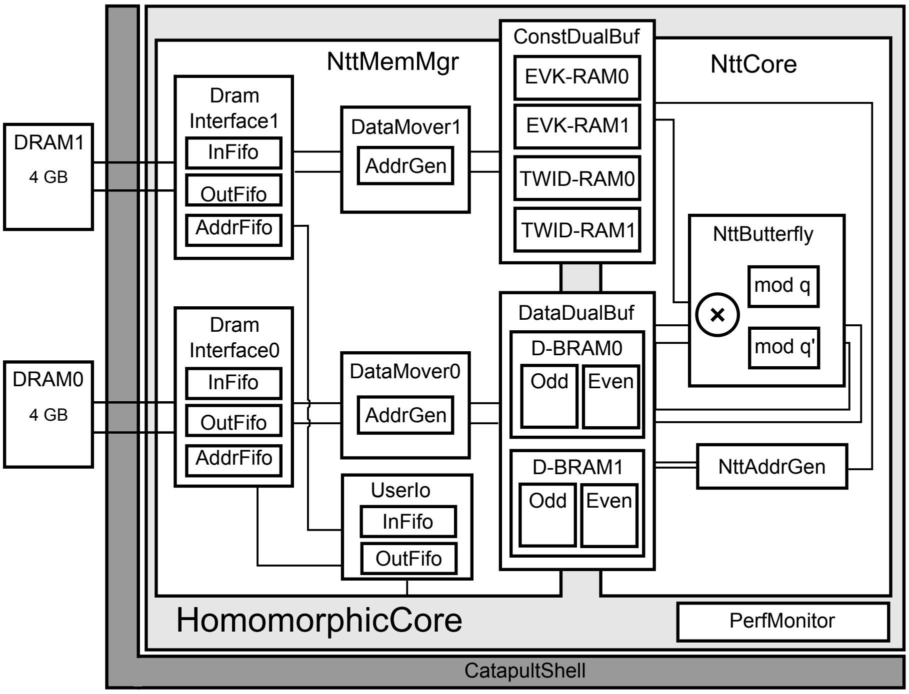

Figure 9.2: Block diagram of our `HomomorphicCore` core used to implement YASHE.

`DataDualBuf`. The `DataDualBuf` buffer contains a configurable number of groups of a polynomial and the `ConstDualBuf` buffer contains the constants (e.g., twiddle factors or evaluation keys) that correspond to the groups in `DataDualBuf`. To the `NttCore` it does not matter which subset of the cached-NTT has to be computed as this is only determined by the loaded data and twiddle factors. This makes the design simpler and also easier to test. To support moduli q and q' we implemented two butterfly units that share one large integer multiplier. Both buffers are double-buffered so that the `NttCore` component can compute on one subset of the data while the `NTTMemMgr` component can load or store a new subset from or into the other buffer. Ciphertexts, NTT constants, and keys are held in one of the two DRAMs (`Dram0` or `Dram1`) and are provided to the core from the outside over the `UserIo` and `CatapultShell` components. The `CatapultShell` component implements a simple PCI Express (PCIe) interface that allows the host server to issue commands (e.g., Add, or Mult) and to transfer data. Evaluated ciphertexts are also stored in the DRAM and can be read by the host after a computation is finished.

9.4 Hardware Architecture

In this section we describe our hardware architecture with an emphasis on the memory bandwidth-friendly cached-NTT polynomial multiplier.

9.4.1 Implementation of the Cached-NTT and Memory Addressing

A crucial aspect when implementing the cached-NTT is efficient access to the main memory (i.e., DRAM) and the use of burst transfers. In this section we describe how data is transferred between the main memory (`Dram0` and `Dram1`) and the cache memory (`DataDualBuf` and `ConstDualBuf`) and how these transfers are optimized.

General Idea

The cached-FFT has been designed for systems with a small cache that supports fast access to coefficients during the computation of a CT-NTT on a group. For our core we do not have a transparent cache, like on a CPU, but implement the fast directly addressable internal on-chip memories `DataDualBuf` and `ConstDualBuf` using BRAMs. As we know exactly which values are required at which time, we explicitly load a group into the internal memory before and write it back after a CT-NTT computation. The necessary reordering (see Figure 9.1) is either performed before or after a computation on a group and done when reading from or writing data into the DRAM. As the DRAM is large enough, plenty of memory is available for temporary storage, but one epoch has to be computed completely and the reordering has to be finished before the next epoch can be computed. In general, it would be sufficient to just store one group consisting of $C = n/G$ coefficients in each buffer of `DataDualBuf`. However, we allow the storage and computation on K groups/caches (configurable as generic during synthesis) in `D-BRAM0` and `D-BRAM1` at the same time (when computing modulo q). One reason is that for relatively small groups we can then avoid frequent waiting for the pipeline to clear after a CT-NTT has been computed. Additionally, storing of multiple groups allows more efficient usage of burst reads and writes.

For efficiency and simplicity we restrict our implementation to a cached-NTT with two epochs[3] (see Table 9.1 for possible numbers of epochs and groups). While it is in general possible to also support three or more epochs this would lead to more memory transfers. The reason is that each epoch requires one to write the whole polynomial subsequently into the cache and also to subsequently read the whole polynomial after the CT-NTT computations are finished. Additionally, more than two epochs result in more complicated address generation. Thus, our implementation supports only dimensions $n = 2^{n'}$ for even values of n'. For Set I we use (n=4096, E=2, G=64, P=6) and for Set II (n=16384, E=2, G=128, P=7).

Supported Commands

To simplify the implementation of homomorphic evaluation algorithms (see Section 9.5) and to abstract away implementation details we support a specific set of instructions to store or load groups or constants and to compute the CT-NTT on such stored groups. A simplified set of available commands is provided in Table 9.2. These commands could also be used to implement other homomorphic schemes and they can be directly used to realize polynomial multiplication in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ and $\mathbb{Z}_{q'}[\mathbf{x}]/(\mathbf{x}^n + 1)$.

Each command consists of a name, which is mapped to an opcode, and zero, one, or two parameters that define the source or destination of data to be transferred or the buffer on

---

[3] With only one epoch the cached-NTT becomes the standard Cooley-Tukey NTT and the cache contains all n coefficients.

which a computation should be performed. A command either blocks `Dram0`, `Dram1`, or `NttCore` and commands can be executed in parallel, in case no resource conflict happens. Memory transfer and computation commands do not interfere due to the dual-buffering. Additionally, commands can be configured for specific situations. For commands operating on `Dram0` or `Dram1` the configuration describes how a storage operation should be performed. Supported modes are a continuous burst transfer ([burst]), or bit-reversal of coefficients ([bitrev]), and/or cached-NTT reordering ([reorder]) during a write or read operation. The $[q]$ and $[q']$ configuration determines whether transfers operate on polynomials modulo q or polynomials modulo q'. When a homomorphic operation has to be performed the top-level state machine also has to provide the base address of the inputs and the base address of the result memory block. Each command also supports a specific maximum burst transfer size, which is discussed in the next paragraph.

The commands themselves are described in Table 9.2. As an example, the load-group[burst]($\mathbf{t}$,x) command loads groups x to $x{+}K{-}1$ from the DRAM at base-address of $\mathbf{t}$ into a buffer using the DRAM's fast burst mode. The store-group[reorder, bitrev]($\mathbf{t}$,x) command stores the groups x to $x{+}K{-}1$ in the DRAM at base-address of $\mathbf{t}$ but performs the reordering of the cached-NTT and also a bit-reversal. An example of a command used to load constants is the load-twiddles[fwd,q](x, y) command that loads the twiddle factors required to compute groups x to $x{+}K{-}1$ in epoch y using burst mode. While the previous commands can be used to implement general polynomial multiplication, we also provide the YASHE specific load-group-expand, load-chunk, and store-chunk commands. The reason is that the KeySwitch algorithm requires the expansion of one polynomial into $\ell_{w,q}$ polynomials (from now on also referred to as *chunks*). For efficiency reasons, the computations are thus performed in parallel on all decomposed polynomials and the larger amount of data to be transferred is handled by the previously mentioned commands. The width of the data ports of `DataDualBuf` and `ConstDualBuf` is $\frac{\lceil \log_2 q' \rceil}{2}$ bits so that we can either store one coefficient modulo q in one position or half of a coefficient modulo q'. As a consequence, the minimal size of `D-BRAM0` and `D-BRAM1` is $\frac{\lceil \log_2 q' \rceil \cdot K \cdot n \cdot \ell_{w,q}}{2G}$ bits.

Usage of Burst Transfers

A significant advantage of storing multiple groups is that this allows the usage of the DRAM's burst mode. In case memory is written or read continuously ([burst]) it is straightforward to see that KC coefficients can be handled in one burst transfer. But also when performing the cached-NTT reordering ([reorder]) the simultaneous reordering of multiple groups allows better utilization of burst operations[4]. In Figure 9.3 the cache can hold $K = 4$ groups and it becomes evident this allows to write K coefficients using burst mode. In this example coefficient 0 (group 0), coefficient 8 (group 1), coefficient 16 (group 2), and coefficient 24 (group 3) will be saved in a continuous memory region after reordering (burst 0). Thus in general, by iterating over the groups and then over the addresses we can write K coefficients using burst mode to reduce memory transfer times significantly. Note that the non-continuous access to memory in `D-BRAM0` does not introduce a performance bottleneck as the memory is implemented using BRAMs that do not cause a performance penalty when being accessed non-continuously.

---

[4] In the following we only discuss the case of writing coefficients from the FPGA (BRAM) into the external memory (DRAM) in reordered or reordered and bit-reversed fashion. However, the same ideas can also be applied when loading from the DRAM and writing into the BRAM on the FPGA.

Table 9.2: Commands that are used to implement YASHE with `HomomorphicCore` where depending on the configuration of each memory transfer command different burst widths can be realized.

<table>
<tr><th>Command</th><th>Param. p_1</th><th>Param. p_2</th><th>Resource</th><th>Configuration</th><th>Coefficients</th><th>Max. burst coefficients</th></tr>
<tr><td>load-group-expand</td><td>DRAM address</td><td>group</td><td><code>Dram0</code></td><td>[burst]</td><td>Kn/G</td><td>Kn/G</td></tr>
<tr><td colspan="7">Loads groups p_2 to $p_2 + K - 1$ using p_1 as base address, performs the decomposition $\mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}) = ([(\tilde{\mathbf{c}}_{\text{mult}})_i]_w)_{i=0}^{\ell_{w,q}-1}$ into $\ell_{w,q}$ polynomials, and stores the decomposed polynomials in the <code>DataDualBuf</code> buffer.</td></tr>
<tr><td>store-chunks</td><td>DRAM address</td><td>group</td><td><code>Dram0</code></td><td>[burst,q]
[burst,q']</td><td>$\ell_{w,q}Kn/G$
$2Kn/G$</td><td>$\ell_{w,q}Kn/G$
$2Kn/G$</td></tr>
<tr><td colspan="7">Saves groups p_2 to $p_2{+}K{-}1$ of all $\ell_{w,q}$ decomposed polynomials ([q]) or spitted coefficients modulo q' ([q']) stored in <code>DataDualBuf</code> at base address p_1.</td></tr>
<tr><td>load-chunks</td><td>DRAM address</td><td>group</td><td><code>Dram0</code></td><td>[burst,q']
[reorder,q']
[reorder,bitrev,q']
[reorder,q]</td><td>$2Kn/G$
$2Kn/G$
$2Kn/G$
$\ell_{w,q}Kn/G$</td><td>$2Kn/G$
$2K$
$2n/G$
$\ell_{w,q}K$</td></tr>
<tr><td colspan="7">Equivalent to store-chunks.</td></tr>
<tr><td>store-group</td><td>DRAM address</td><td>group</td><td><code>Dram0</code></td><td>[burst]
[reorder,bitrev]
[reorder]</td><td>Kn/G
Kn/G
Kn/G</td><td>Kn/G
n/G
K</td></tr>
<tr><td colspan="7">Saves groups p_2 to $p_2{+}K{-}1$ of the polynomial stored in <code>DataDualBuf</code> at base address p_1.</td></tr>
<tr><td>load-group</td><td>DRAM address</td><td>group</td><td><code>Dram0</code></td><td>[burst]
[bitrev]</td><td>Kn/G
Kn/G</td><td>Kn/G
1</td></tr>
<tr><td colspan="7">Equivalent to store-group.</td></tr>
<tr><td>load-twiddles</td><td>group G</td><td>epoch E</td><td><code>Dram1</code></td><td>[(fwd|inv),q]
[(fwd|inv),q']</td><td>$n/(2G)$
n/G</td><td>$n/(2G)$
n/G</td></tr>
<tr><td colspan="7">Loads the precomputed forward or inverse twiddle factors for modulus q or q' for groups p_1 to $p_1 + K$ and epoch $E = p_2$ into <code>ConstDualBuf</code> using burst read.</td></tr>
<tr><td>load-psis</td><td>group G</td><td>-</td><td><code>Dram1</code></td><td>[q]
[q']</td><td>n/G
$2n/G$</td><td>n/G
$2n/G$</td></tr>
<tr><td colspan="7">Loads the powers of ψ^{-1} for groups p_1 to $p_1 + K - 1$ and moduli q or q' from DRAM using burst read and saves them in <code>ConstDualBuf</code>.</td></tr>
<tr><td>load-evks</td><td>DRAM address</td><td>group</td><td><code>Dram1</code></td><td>-</td><td>$\ell_{w,q}n/G$</td><td>$\ell_{w,q}n/G$</td></tr>
<tr><td colspan="7">Loads the $\ell_{w,q}$ different evaluation key parts for groups p_2 to $p_2 + K - 1$ stored at base address p_1 into <code>ConstDualBuf</code> using burst read. Evaluation keys are always modulo q.</td></tr>
<tr><td>ntt-on-buffer</td><td>chunk</td><td>-</td><td><code>NttCore</code></td><td>[(q|q')]</td><td>-</td><td>-</td></tr>
<tr><td colspan="7">Computes the CT-NTT on chunk p_1 stored in <code>DataDualBuf</code> using either modulus q or modulus q' and requiring $\frac{Pn}{2G}$ multiply-accumulate (MAC) operations.</td></tr>
<tr><td>mul-psi</td><td>chunk</td><td>-</td><td><code>NttCore</code></td><td>[q]
[q',round]</td><td>-
-</td><td>-
-</td></tr>
<tr><td colspan="7">Multiplies chunk p_1 stored in <code>DataDualBuf</code> by powers of ψ^{-1} stored in <code>ConstDualBuf</code>. If configured with [round] the YASHE rounding operation is performed after the NTT.</td></tr>
<tr><td>mul-evk</td><td>chunk</td><td>-</td><td><code>NttCore</code></td><td>-</td><td>-</td><td>-</td></tr>
<tr><td colspan="7">Multiplies chunk p_1 in <code>DataDualBuf</code> by the evaluation keys stored in <code>ConstDualBuf</code>. Operates only on polynomials modulo q.</td></tr>
<tr><td>accumulate</td><td>chunk</td><td>-</td><td><code>NttCore</code></td><td>-</td><td>-</td><td>-</td></tr>
<tr><td colspan="7">Adds chunks p_1 to chunk 0 stored in <code>DataDualBuf</code>. Operates only on polynomials modulo q.</td></tr>
<tr><td>mul-point-wise</td><td>-</td><td>-</td><td><code>NttCore</code></td><td>[(q|q')]</td><td>-</td><td>-</td></tr>
<tr><td colspan="7">Point-wise multiplication of two polynomials in <code>ConstDualBuf</code>.</td></tr>
</table>

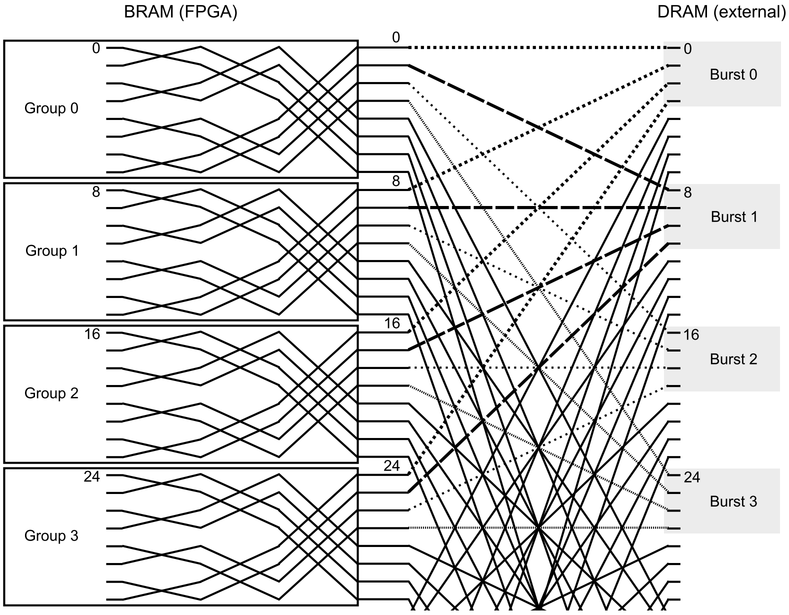

Figure 9.3: Usage of burst mode when performing the reordering when writing coefficients from the internal buffer to the external DRAM for a cached-NTT with parameters ($n = 64$, $E = 2$,$G = 8$) and $K = 4$.

Another improvement is visualized in Figure 9.4 which shows the combination of the bit-reversal with the reordering procedure of the cached-NTT ([reorder,bitrev]). When computing BitrevAddr(Reorder(addr)) it becomes evident that the content of each group can be transferred continuously. It is thus possible to write a whole group ($C = n/G$ coefficients) using burst mode. In this case it is only necessary to read the coefficients in non-continuous order from the BRAM and to compute an offset to store them using burst mode. From this analysis it can be seen that it is even preferable to compute the reordering together with the bit-reversal instead of only the reordering, as the size of the burst write is even larger in this case for relevant parameters (i.e., n/G instead of G).

9.4.2 Computation of the CT-NTT on the Cache

The CT-NTT is computed on each group in the cache (see the black box in Figure 9.1) and requires arithmetic operations that dominate the area costs of our implementation. Each CT-NTT on a group requires $\frac{Pn}{2G}$ multiplications in $\mathbb{Z}_q$ (or $\mathbb{Z}_{q'}$) and the whole cached-NTT requires $EG\frac{Pn}{2G} = \frac{n\log_2(n)}{2}$ multiplications in $\mathbb{Z}_q$ (or $\mathbb{Z}_{q'}$). The number of stages or passes of the CT-NTT, which are the recursive divisions into sub-problems, is $P = \log_2(n/G)$. The address generation in `NttCore`, which implements the CT-NTT, is independent of the group or epoch that is processed. This allows a simple data-path and also testability independently of the memory transfer commands. To saturate the pipelined butterfly unit of the NTT, two reads and two writes are required per cycle and we use the well-known fact that the buffer can be split into two memories, one for even and one for odd addresses (see [Pea68]). While this

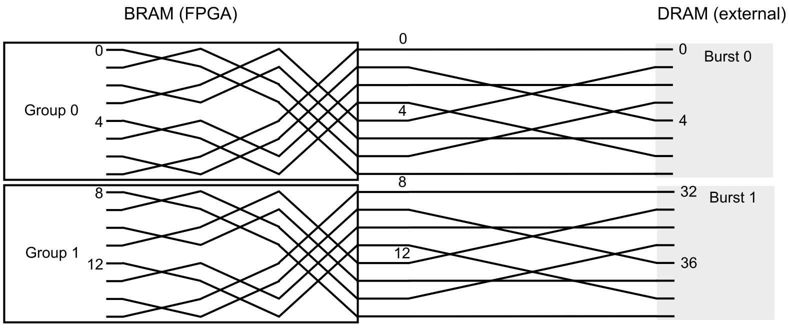

Figure 9.4: Usage of burst transfers between the internal cache (BRAM) and the main memory (DRAM) with cached-NTT parameters ($n = 64$, $E = 2$, $G = 8$) and a memory transfer command using [reorder,bitrev].

approach might lead to wasted space in block memories if small polynomials do not fill a whole block RAM, as in [PG12] and optimized in [APS13, RVM⁺14], it is not a concern for the large parameter sets we are dealing with. The only input to the NTT, besides the actual polynomial coefficients, that depend on the current group or epoch, are the constants like twiddle factors, powers of ψ^{-1}, or the evaluation key evk. We decided to store each constant in a continuous memory region and load them into the `TWID-RAM` or `EVK-RAM` buffers depending on the current group or epoch. While it would also be possible to compute the twiddle factors on-the-fly (as in [RVM⁺14]) this approach would require an additional expensive $q' \times q'$ multiplier and modulo unit. Additionally, we do not exploit redundancies in twiddle factors or other tricks so that we are able to load constants using the fast burst mode. The only important observation is that when $E = 2$ the same twiddle factors are used for the computation of all groups of the first epoch of the NTT.

Large Integer Multiplication

For best performance of the NTT_q, our architecture requires a pipelined NTT butterfly that is able to compute a $\log_2(q) \times \log_2(q)$ multiplication, modular reduction, and two accumulations per cycle. For the butterfly of the $\mathrm{NTT}_{q'}$, execution in one clock cycle is not necessary as the maximum data width of the `ConstDualBuf` and `DataDualBuf` components is $\frac{\lceil \log_2(q) \rceil}{2}$. Thus at least two cycles are needed to load a coefficient from the buffer in which one coefficient modulo q' is split into chunk 0 and chunk 1. For the design of the $\log_2(q) \times \log_2(q)$ multiplier we tried to be flexible enough for eventual changes of parameters and future extension and designed two 576×576-bit multipliers matching the range of the 576-bit DRAM interface so that we eventually could support KeySwitch only with $\lceil \log_2(q) \rceil = 576$. Currently, the largest required bit width is $\frac{1040}{2} \times \frac{1040}{2}$-bits, when the multiplier is used in time-shared mode to implement a pipelined $q' \times q'$-bit multiplication in four cycles.

To instantiate the multiplier we used a traditional RTL design ($\mathsf{MUL}_{\mathrm{RTL}}$) that uses four pipelined 72×72-bit multipliers generated using the Altera MegaWizard to instantiate a 144×144-bit multiplier. The instantiation of four 144×144-bit multipliers yields a 288×288-bit multiplier and finally a pipelined 576×576-bit multiplier. We also investigated whether the

DSP Builder Advanced Blockset, which runs within the Simulink environment, could be used to design a large integer multiplier. A standalone instantiation of a multiplication core ($\mathsf{MUL}_{\mathrm{DSP}}$) using DSP Builder allowed higher clock rates than $\mathsf{MUL}_{\mathrm{RTL}}$ but when we used $\mathsf{MUL}_{\mathrm{DSP}}$ in our core the synthesis and fitting time increased and final resource consumption was unchanged.

Solinas Reduction

For modular reduction common methods are Barrett reduction [Bar86], Montgomery reduction [Mon85], and reductions based on special prime numbers [VOMV96]. However, for very large values, generic methods like Barrett or Montgomery modular reduction require a large number of additions or even a large multiplier and are thus expensive, slow, and also hard to write generically. As a consequence, we restrict the moduli q and q' to Solinas primes [Sol99] of the form $2^y - 2^z + 1$ for $y, z \in \mathbb{Z}$ and $y > z$. A modular reduction circuit can then be configured by providing the input bit width and the values y and z as generics. The implementation only requires a few shifts and few additions/subtractions to perform a modular reduction. To achieve timing a sufficient number of registers has been inserted between adders and shifters in the RTL design.

9.5 Configuration of our Core for YASHE

For our prototype we have implemented YASHE's homomorphic evaluation operations Add and Mult using the architecture described in Section 9.4. For brevity, we only cover the RMult and KeySwitch functions in detail, which are essential for the implementation of Mult. All homomorphic evaluation operations use the hardware architecture described in Section 9.4 and the commands provided in Table 9.2. The commands are executed by a large state machine implemented in `HomomorphicCore`, which is also responsible for interaction with the Catapult shell and host PC.

9.5.1 Implementation of RMult

For RMult, a standard integer polynomial multiplication in $\mathbb{Z}_{q'}[\mathbf{x}]/(\mathbf{x}^n+1)$ is required after which the result is rounded and reduced modulo q. Selecting $q' > nq^2$ guarantees that the product $\mathbf{c}_1\mathbf{c}_2$ of two polynomials $\mathbf{c}_1, \mathbf{c}_2 \in \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ is computed over the integers and not being reduced before it is rounded. Instead of using a single routine for RMult, the host server can make separate calls to a single forward transformation $\bar{\mathbf{c}}_i = \mathsf{RMultFwd}(\mathbf{c}_i)$ so that polynomials to be multiplied by multiple other polynomials have to be transformed only once into the NTT domain. The $\tilde{\mathbf{c}}_{\mathrm{mult}} = \mathsf{RMultInv}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2)$ routine then takes two transformed polynomials $\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2$ as input and computes the product by performing point-wise multiplication, the inverse NTT, and rounding of the result. While we give up some efficiency (e.g., merging of forward transformation and point-wise multiplication) by this approach, it seems beneficial to provide this additional flexibility when computing homomorphic circuits.

The (simplified) sequence of executed commands for RMultFwd is provided in Algorithm 35, but for the actual implementation load/store operations and NTT computations are executed in parallel to make use of the double-buffer capability of the `DataDualBuf` and `ConstDualBuf` components. In step 5 of RMultFwd the input polynomial is expected to be saved in bit-reversed order already. This is either ensured by the user when the polynomial is initially transferred to the device or by our implementation in the last step of KeySwitch. The only execution of a reordering load operation appears in step 11 and all other loads or stores use the burst mode. Thus the second reordering is delayed till the pointwise multiplication in RMultInv which is given in Algorithm 36. In RMultInv the first block of operations (step 3 to 7) is responsible for the pointwise multiplication. Note that the Add operation of YASHE is basically this loop but mul-point-wise is exchanged by a command for addition in $\mathbb{Z}_q$. The first NTT-related load is performed in step 11 in which the final reordering of the forward transform together with the bitreversal step is performed. The final rounding operation $\left[\left\lfloor \frac{t}{q}\mathbf{t}_2 \right\rceil\right]_q$ is included into the mul-psi[q', round] command. After that the result $\tilde{\mathbf{c}}_{\text{mult}}$ is in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$.

Algorithm 35 Forward Transformation in RMult

```
 1: func RMultFwd($\mathbf{c}_i$)
 2:   //Epoch 0
 3:   load-twiddles[fwd,$q'$]$(0, 0)$
 4:   forall groups $x \in 0 \ldots G/K - 1$:
 5:       load-group[burst]($\mathbf{c}_i$, $Kx$)
 6:       ntt-on-buffer[$q'$](0)
 7:       store-chunks[burst,$q'$]($\mathbf{t}$, $Kx$)
 8:   //Epoch 1
 9:   forall groups $x \in 0 \ldots G/K - 1$:
10:       load-twiddles[fwd,$q'$]$(Kx, 1)$
11:       load-chunks[reorder,$q'$]($\mathbf{t}$, $Kx$)
12:       ntt-on-buffer[$q'$](0)
13:       store-chunks[burst,$q'$]($\bar{\mathbf{c}}_i$, $Kx$)
14:   return $\bar{\mathbf{c}}_i$
15: end func
```

Algorithm 36 Pointwise Multiplication and Inverse Transformation in RMult

```
 1: func RMultInv($\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2$)
 2:   //Pointwise multiplication
 3:   forall groups $x \in 0 \ldots G/K - 1$:
 4:       load-chunks[burst,$q'$]($\mathbf{c}_1$, $Kx$)
 5:       load-chunks[burst,$q'$]($\mathbf{c}_2$, $Kx$)
 6:       mul-point-wise[$q'$]()
 7:       store-chunks[burst,$q'$]($\mathbf{t}_1$, $Kx$)
 8:   //Epoch 0
 9:   load-twiddles[inv,$q'$]$(0, 0)$
10:   forall groups $x \in 0 \ldots G/K - 1$:
11:       load-chunks[reorder,bitrev,$q'$]($\mathbf{t}_1$, $Kx$)
12:       ntt-on-buffer[$q'$](0)
13:       store-chunks[burst,$q'$]($\mathbf{t}_2$, $Kx$)
14:   //Epoch 1
15:   forall groups $x \in 0 \ldots G/K - 1$:
16:       load-twiddles[inv,$q'$]$(Kx, 1)$
17:       load-psis[$q'$]$(Kx)$
18:       load-chunks[reorder,$q'$]($\mathbf{t}_2$, $Kx$)
19:       ntt-on-buffer[$q'$](0)
20:       mul-psi[$q'$, round](0)
21:       store-group[reorder,bitrev]($\tilde{\mathbf{c}}_{\text{mult}}$, $Kx$)
22:   return $\tilde{\mathbf{c}}_{\text{mult}}$
23: end func
```

9.5.2 Implementation of KeySwitch

The control-flow used to implement KeySwitch based on the commands introduced in Section 9.4 and Equation 9.1 is given in Algorithm 37. For the forward transformation (step 2 to step 19) the coefficients of the input polynomial $\tilde{\mathbf{c}}_{\text{mult}}$ can be loaded using the burst mode as they have already been stored in bitreversed representation in RMultInv. The decomposition $\mathsf{Dec}_{w,q}(\tilde{\mathbf{c}}_{\text{mult}}) = ([(\tilde{\mathbf{c}}_{\text{mult}})_i]_w)_{i=0}^{\ell_{w,q}-1}$ is performed on-the-fly inside the FPGA using the load-group-expand[burst] command. The NTT is then performed on all $\ell_{w,q}$ decomposed polynomials in the buffer. As the twiddle factors are identical for each polynomial we only have to load and store K sets of twiddle factors into the `ConstDualBuf` component (each set containing $P \cdot \ell_{w,q}/2$ coefficients). During the NTT computation on all polynomials the results are accumulated (step 18) and then stored (step 19). The relatively slow reordering operation load-chunks[reorder, q] is performed at the beginning of the second epoch and not after the first epoch as the accumulation and multiplication with the evaluation keys takes additional time so that we can balance the time required for memory transfers and computation. As the forward transformed polynomials are already stored in the correct order, we just have to perform a burst read at the beginning of the inverse transformation in step 24. Additionally, the computation is much less involved as we only have to compute one INTT_q and not $\ell_{w,q}$ computations of NTT_q caused by the decomposition. It is not possible to merge the multiplication by powers of ψ^{-1} into the NTT twiddle factors for the inverse transformation [RVM+14] as we use the Cooley-Tukey butterfly (see Section 6.2). The multiplication by powers of ψ^{-1} is performed by the mul-psi command and the constants are loaded into the memory space reserved for the evaluation key during the forward transformation by load-psis. The multiplication by the scalar n^{-1} is merged into the ψ^{-1} values.

Algorithm 37 Key Switching in YASHE

```
 1: func KeySwitch($\tilde{\mathbf{c}}_{\text{mult}}, \overline{\mathsf{evk}}$)
 2:   //Fwd. transform and accumulation:
 3:   load-twiddles[fwd,$q$]$(0,0)$
 4:   //Epoch 0
 5:   forall groups $x \in 0 \dots G/K - 1$:
 6:       load-group-expand[burst]$(\tilde{\mathbf{c}}_{\text{mult}}, Kx)$
 7:       forall chunks $y \in 0 \dots \ell_{w,q} - 1$:
 8:           ntt-on-buffer[$q$]$(y)$
 9:       store-chunks[burst,$q$]$(\mathbf{t}_1, Kx)$
10:   //Epoch 1
11:   forall groups $x \in 0 \dots G/K - 1$:
12:       load-twiddles[fwd,$q$]$(Kx, 1)$
13:       load-evk$(\overline{\mathsf{evk}}, Kx)$
14:       load-chunks[reorder, $q$]$(\mathbf{t}_1, Kx)$
15:       forall chunks $y \in 0 \dots \ell_{w,q}$:
16:           ntt-on-buffer[$q$]$(y)$
17:           mul-evk[$q$]$(y)$
18:           accumulate$(y)$
19:       store-group[reorder, bitrev]$(\mathbf{t}_2, Kx)$
20:   //Inverse transform:
21:   load-twiddles[inv,$q$]$(0,0)$
22:   //Epoch 0
23:   forall groups $x \in 0 \dots G/K - 1$:
24:       load-group[burst]$(\mathbf{t}_2, Kx)$
25:       ntt-on-buffer[$q$]$(0)$
26:       store-group[reorder]$(\mathbf{t}_1, Kx)$
27:   //Epoch 1
28:   forall groups $x \in 0 \dots G/K - 1$:
29:       load-twiddles[inv,$q$]$(Kx, 1)$
30:       load-psis[$q$]$(Kx)$
31:       load-group[burst]$(\mathbf{t}_1, Kx)$
32:       ntt-on-buffer[$q$]$(0)$
33:       mul-psi[$q$]$(0)$
34:       store-group[reorder, bitrev]$(\mathbf{c}_{\text{mult}}, Kx)$
35:   return $\mathbf{c}_{\text{mult}}$
36: end func
```

9.6 Results and Comparison

In this section we provide post place-and-route (post-PAR) results and performance measurements of our implementation on the Catapult board [PCC+14] equipped with an Altera Stratix V (5GSND5H) FPGA and two 4 GB DRAMs.

9.6.1 Resource Consumption and Performance

The resource consumption of our implementation is reported in Table 9.3. Achieving a high clock frequency for parameter Set II is challenging. One reason seems to be that, due to our design choices, we have to deal with extremely large structures like several thousand bit wide adders and a large integer multiplier. Such structures are tedious to manually optimize and it is hard to determine an optimal pipeline length. Another reason is that the design is congested and that placement and fitting have to satisfy strict constraints imposed by the PCIe and DRAM controllers in the Catapult shell. Still, switching to larger devices to reduce congestion would also increase costs. In our core the critical path is currently in the pipelined rounding circuit required for RMult and we instantiated our design using the $\mathsf{MUL}_{\mathrm{RTL}}$ multiplier which allows faster simulation and synthesis compared to the $\mathsf{MUL}_{\mathrm{DSP}}$ design provided by the DSP Builder. However, higher clock frequencies in future work might be easier to achieve with automated tools like the DSP Builder and the resulting $\mathsf{MUL}_{\mathrm{DSP}}$ design.

Table 9.3: Resource consumption of our implementation of YASHE (including the communication interface).

Implementation	ALM	FF	DSP	BRAM bits	MHz
Set I (n=4096, K= 8)	69,058 (40 %)	144,747	144 (9 %)	8,031,568 (19 %)	100
Set II (n=16384, K=4)	141,090 (82 %)	391,773	577 (36 %)	17,626,400 (43 %)	66

Cycle counts for evaluation operations are given in Table 9.4 and are obtained using the `PerfMonitor` component that logs cycle counts and transfers them to the host server over PCIe, if requested. The usual approach of obtaining cycle counts from simulation is not possible as we are using an external DRAM without a cycle accurate simulation model. Moreover, the runtime does not simply scale for higher clock frequencies as the memory interface is running in its own clock domain and thus the memory bandwidth is not significantly increased by higher clock frequencies of the `HomomorphicCore` component.

A good indicator for the efficiency of our memory addressing is the saturation of the $\log(q) \times \log(q)$ modular multiplier. One NTT requires $\frac{n}{2}\log_2(n)$ multiply-accumulate (MAC) operations so that KeySwitch operating on G groups and two epochs takes at least $C_{KS}(\ell_{w,q}, n) = (\ell_{w,q} + 1)(\frac{n}{2}\log_2(n) + n)$ cycles assuming one clock cycle per MAC ($\ell_{w,q}$ forward and one inverse NTT, see Equation 9.1). For parameter Set II we get $C_{KS}(8, 16384) = 1{,}179{,}648$ as lower bound on the number of cycles for KeySwitch which is close to the measured 1,372,519 cycles. For RMult approx. $C_{\mathrm{RM}}(n) = 3(4\frac{n}{2}\log_2 n) + 2(4n)$ cycles are required (three transformations, point-wise and ψ^{-1} multiplication; four cycles per MAC) and the saturation of the MAC unit is $\frac{C_{\mathrm{RM}}(16384)}{1{,}839{,}987} = 0.84$.

Table 9.4: Cycle counts and runtimes for the different evaluation algorithms of YASHE measured on the Catapult board.

Implementation		Mult	Add	KeySwitch	RMult	RMultFwd	RMultInv
Set I (n=4096)	cycles	675,326	19,057	478,911	196,415	160,693	157,525
100 MHz (K=8)	time	6.75 ms	0.19 ms	4.79 ms	1.96 ms	1.61 ms	1.58 ms
Set II (n=16384)	cycles	3,212,506	61,775	1,372,519	1,839,987	587,664	664,659
66 MHz (K=4)	time	48.67 ms	0.94 ms	20.80 ms	27.88 ms	8.90 ms	10.07 ms

9.6.2 Comparison with Previous Work

Cao et al. [CMO+14] describe an implementation of the integer-based FHE scheme by Coron et al. [CMNT11] on a Virtex-7 FPGA (XC7VX980T) but unlike our work they explicitly do not take into account the bottleneck that may be caused by accessing off-chip memory. The integer multiplication is realized with the integer-FFT algorithm from [EW11] and a large Barrett reducer that uses Virtex-7 DSPs. Their implementation achieves a speed-up factor of 11.25 compared to a software implementation but for large parameter sets, which might promise even higher speed-ups (e.g., 44), the design does not fit on current FPGAs anymore and only synthesis results are given. An FPGA implementation of an integer multiplier for the Gentry-Halevi [GH11] FHE scheme is proposed by Wang and Huang in [WH13]. The architecture requires about 462,983 ALUs, and 720 DSPs on a Stratix-V (55GSMD8N3F45I4) and allows 768K-bit multiplications by using a 64k-point FFT. It is reported to be about two times faster than an implementation on an NVIDA C2050 graphics processing unit (GPU) using a similar approach. Another 768K-bit multiplication architecture was proposed by Wang et al. in [WHEW14] targeting ASICs and FPGAs. An outline of an implementation of a homomorphic encryption scheme is given in [CRPS12] using Matlab/Simulink and the Mathwork HDL coder. It uses the Chinese remainder theorem (CRT) and the NTT but the used tools limit the available basic multiplier width to 128 bits and the design approach would require multiple FPGAs in order to deal with large vector lengths up to $n = 2^{14}$.

An ASIC implementation of a million-bit multiplier for integer-based FHE schemes has been presented by Doröz et al. in [DÖS13]. It uses the Schönhage-Strassen algorithm and the NTT, which operates on an on-chip cache. The computation of the product of two 1,179,648-bit integers takes 5.16 million clock cycles. Synthesis results for a chip using the TSMC 90 nm cell library show a maximum clock frequency of 666 MHz and thus a runtime of 7.74 ms for this operation. The whole chip requires 26.7 million gates where 26.5 million gates are attributed to the 768 Kbyte cache and the runtime is equivalent to a software implementation. This shows, similar to our result, that the biggest challenges in the implementation of homomorphic cryptography in hardware are the large ciphertexts that do not fit into block RAMs (our case) or caches instantiated with the standard library (Doröz et al. [DÖS13]).

Wang et al. [WHC+12] presented the first GPU implementation of an FHE scheme and provide results for the Gentry-Halevi [GH11] scheme on an NVIDIA C2050 GPU. The results were subsequently improved in [WHC+15] as all computations are carried out in the FFT-domain. A GPU implementation of the leveled fully homomorphic BGV scheme [BGV12] is given in [WCH14].

In more recent work [DÖS15] Dai et al. provide an implementation of the Doröz-Hu-Sunar (DHS) [DHS14] NTRU-based fully homomorphic scheme based on LTV [LTV12]. For the parameter set ($n = 16384, \log(q) = 575$) that supports the evaluation of the 24 level deep decryption circuit of the Prince block cipher [BCG+12], they require 0.063 seconds for multiplication and 0.89 seconds for relinearization (key switching) on a 2.5 GHz Xeon E5-2609 equipped with an NVIDIA GeForce GTX 690. While the parameters for polynomial arithmetic and the DHS scheme are similar to YASHE, in DHS one basically sets $\ell_{w,q} = \lceil \log_2(q) \rceil$ (i.e., the evaluation key consists of $\lceil \log_2(q) \rceil$ polynomials) and thus far more forward transformations are required for key switching. However, their reported performance of RMult is almost the same as in our implementation.

A software library that implements the BGV [BGV11, BGV12] scheme is described in [HS14] and freely available. In [LN14], a software implementation of YASHE is reported which for the parameter set ($n = 4096, q = 2^{127} - 1, w = 2^{32}$) executes Add in 0.7 ms, RMult in 18 ms, and KeySwitch in 31 ms on an Intel Core i7-2600 running at 3.4 GHz. Thus our hardware implementation can evaluate Mult on a parameter set supporting 9 levels in 48.67 ms while a software implementation requires 49 ms for a parameter set supporting only 1 multiplicative level.

In concurrent work Roy et al. [RJV+15] proposed an implementation of YASHE with parameter $n = 2^{15}$ and a modulus of $\log_2(q) = 1228$ bits. Because of the larger parameter set they can support evaluation of the decryption circuit of the SIMON-64/128 block cipher. Moreover, their implementation does not require to set $\mathbf{f}(\mathbf{x}) = \mathbf{x}^n + 1$ when operating in the polynomial ring $\mathbb{Z}_q[\mathbf{x}]/\langle \mathbf{f} \rangle$ so that single instruction multiple data (SIMD) operations on homomorphic ciphertexts are supported. On the other hand they also use a much larger next generation FPGA (Virtex-7 XC7V1140T) from a different vendor so that a comparison is naturally hard - especially regarding the economic benefits of using FPGAs. We see the biggest contribution of the work by Roy et al. in their efficient implementation of independent processors that use the CRT to decompose polynomials. This approach avoids large integer multiplier and simplifies routing and performance tuning. When we designed our core, the added complexity and the need to lift polynomials from CRT to natural representations in hardware appeared to be too expensive. However, the authors of [RJV+15] do not consider the costs of moving data between external memory and the FPGA but just assume unlimited memory bandwidth. This naturally simplifies the design and placement but does not appear to be a realistic assumption. Especially, when taking into account that DRAM or PCIe cores will also require logic resources on the device, occupy clock domains, and could cause timing problems. In our work a considerable amount of time was spent to implement efficient memory transfers and to optimize the algorithms in this regard. However, we see our work and the work of Roy et al. as a first step towards an efficient accelerator.

All in all, currently not enough data points exist (also due to the vast amount of different schemes) to decide whether GPUs, ASICs, or FPGAs are the most suitable platform for FHE/SHE accelerators. However, huge area costs for caches and long design and manufacturing times do not favor ASICs. FPGAs and GPUs (based on [DHS14]) appear to achieve similar performance. Given that datacenters are a prime candidate for homomorphic encryption, the advantages in scale and total cost of ownership described in [PCC+14] suggest that FPGAs might not only be competitive with GPUs, but even preferable.

9.6.3 Software Performance

Our software implementation of YASHE was mainly written to generate test vectors and to prototype and test the optimized algorithms and in Table 9.5 we provide performance numbers. However, the implementation is not optimized for the CPU architecture but resembles the execution flow of the hardware implementation. A direct comparison with our hardware implementation does not seem fair but we still provide the results to give a rough estimate on performance in software.

We also evaluated RMult realized with Nussbaumer's method [Nus82, BCNS15] (denoted RMult_{Nb}) and obtained better performance than with the NTT using q'. But as a fast software implementation is not in the scope of this work we did not investigate the root cause for the better performance but consider a detailed comparison of both algorithms as valuable future work. In Table 9.7 we also report implementation results of an open-source implementation by Lepoint and Naehrig [LN14]. Their implementation has received more optimization efforts, performs better, but is also benchmarked on a faster CPU architecture.

Table 9.5: Software performance of our prototype implementation of the YASHE evaluation operations as described in Section 9.5.

Parameter Set	RMult_{NTT}	RMult_{Nb}	KeySwitch	Add	RMultFwd	RMultInv
Set I ($n = 4096, \ell_{w,q} = 2$)	190 ms	83 ms	70 ms	0.24 ms	76 ms	75 ms
Set II ($n = 16384, \ell_{w,q} = 8$)	6.21 s	1.73 s	5.04 s	0.0013 s	1.62 s	2.34 s

Experiments were executed on an Intel Core i7-2760 QM CPU running at 2.4 GHz with access to 8 GB RAM.

Table 9.7: Software performance of an implementation of YASHE obtained from [LN14].

Parameter Set	Gen	Encrypt	Decrypt	Mult	KeySwitch	Add
($n = 4096, \ell_{w,q} = 4$)	3.4 s	16 ms	15 ms	18 ms	31 ms	0.7 ms

Experiments were performed using an Intel Core i7-2600 at 3.4 GHz with hyper-threading turned off and over-clocking ('turbo boost') disabled.

9.7 Conclusion and Future Work

In this work we have shown the potential of FPGAs to accelerate somewhat homomorphic encryption despite the large size of ciphertexts and keys. We provided a generically applicable polynomial multiplier and an implementation of the evaluation steps of the YASHE scheme. Our evaluation shows a speedup of roughly 100 times over the software implementation provided in [LN14] or in other words we can support a $n = 16384$ parameter set (roughly 9 levels) in hardware with a running time equivalent to a $n = 4096$ parameter set in software (supporting only one level).

While implementing the scheme we encountered several challenges that might also be a good opportunity for future work. A big issue was verification and simulation time due to the large problem sizes. While parameter Set I can be verified in several minutes it takes more than half an hour to verify Mult and Add for parameter Set II on a standard desktop computer using Questasim 10.4. As a consequence, it is extremely important to develop an implementation using generic structures so that verification and debugging can be performed on small and fast to simulate parameter sets. Additionally, we were not able to build a fully pipelined 1040x1040 multiplier that fits onto the target device. While the amount of DSPs would theoretically be sufficient to construct such a multiplier, we ran out of ALUs for internal adders in our experiments. In general the large parameter sizes in Set II were also challenging, as the design of pipelined arithmetic was time consuming due to long synthesis cycles.

Possible future work is further design space exploration and implementation of even larger parameter sets. Moreover, it might make sense to investigate the applicability of the CRT in combination with the cached-NTT. Another interesting future direction for better FPGA utilization and removal of bottlenecks caused by external memory might be the design of a custom board with one or more suitable FPGAs connected to a maximum amount of external memory units. A huge performance boost for SHE/FHE schemes would also be possible if the costly rounding and non-modulo q multiplication could be removed or if all operations could be carried out directly in the frequency/NTT domain.

Chapter 10

Conclusion and Future Work

In this chapter we provide a short conclusion and summarize the results of this thesis. Moreover, we cover possible areas of further research on lattice-based cryptography. Especially, work dealing with the protection against physical attacks appears to be necessary before lattice-based cryptography can be adopted in practice. Additionally, improved implementations of polynomial multiplication or Gaussian sampling would be beneficial for a wide range of schemes. Another field of further research is the implementation of advanced schemes like identity-based encryption (IBE), attribute-based encryption (ABE), and homomorphic encryption.

Contents of this Chapter

10.1 Conclusion

In this thesis we have predominately discussed promising ideal lattice-based public-key encryption and digital signature schemes but also covered an advanced homomorphic encryption scheme. Our results show that schemes like RLWEenc, GLP, and BLISS are remarkably fast on reconfigurable hardware, constrained devices, and also general purpose microprocessors. Additionally, we have examined a wide range of algorithms for polynomial multiplication (NTT, Karatsuba, schoolbook) and Gaussian sampling (Knuth-Yao, CDT, Ziggurat, or Bernoulli). By exploring the specific advantages of these algorithms we provide guidelines to designers and engineers how to choose an algorithm under specific implementation constraints (e.g., area or performance). Moreover, our implementations of the microcode engine, RLWEenc, and BLISS are freely available for external verification of our results and we see the published source code as an additional contribution of this thesis. Our work also shows that lattice-based public-key encryption can be optimized for high-performance or low-area and that it could be an interesting alternative to RSA or ECC, especially in cost-sensitive application scenarios. A specific advantage of lattice-based cryptography over other post-quantum cryptoschemes is the possibility to realize public-key encryption and signature schemes using very similar base arithmetic and building blocks. Additionally, the performance impact of high-security parameter sets appears to be much less severe for RLWEenc and BLISS than for RSA or ECC. All in all, in this thesis we were able to show that ideal lattice-based public-key encryption and digital signature schemes are efficient and fast on a wide range of devices.

10.2 Directions for Future Research

Specific suggestions for future work related to individual parts of the contribution of this thesis can be found in the corresponding chapters. However, there are also some main lines of research that appear to be worth further efforts.

Cryptanalysis and Protection against Physical Attacks

Currently, there are presumably two main issues that prevent the wide adoption of lattice-based cryptography. One is certainly lack of trust into the underlying assumptions and the security of current parameter sets. We hope that further cryptanalysis, which is certainly required, will clear up this picture. To provide a true post-quantum alternative it will also be necessary to consider the impact of quantum computers on the cryptanalysis of lattice-based cryptography. While quantum computers might not completely break lattice-based cryptography, they could probably be used to accelerate sub-routines of cryptanalytic algorithms. This does not appear to be taken into account when parameters were selected for current schemes. If lattice-based cryptography passes its mandatory test of time, we are confident that standardization efforts will follow. It might also turn out that only standard lattice-based schemes remain secure while their ideal lattice-based counterparts get broken. Thus, future work should also focus more on the efficiency of standard lattices as fallback. Additionally, for applications that require long term security and that are not constrained by large key sizes, standard lattices currently appear to be a safer way to proceed. New techniques and approaches to deal with large public keys might also help to increase the efficiency of standard lattice constructions on constrained devices.

The second obstacle is the lack of research on physical protection and side-channel countermeasures tailored to lattice-based cryptosystems. In most practical applications at least protection against timing side-channel attacks is mandatory. So far there has been very little research conducted on the vulnerabilities of lattice-based cryptographic implementations to physical attacks (a first work is [RRVV14]). It is anticipated that there may be particular vulnerabilities with respect to algorithms with variable runtime; for instance Gaussian and rejection sampling, which are major components of many lattice-based cryptosystems.

Improved Polynomial Arithmetic and Gaussian Sampling

Polynomial arithmetic and Gaussian sampling are key components used in almost all practical ideal lattice-based cryptoschemes (e.g., BLISS, RLWEenc, or BG). Although polynomial multiplication for ideal lattice-based cryptography has been researched quite well in works like [APS13, RVM+14], it might still be possible to optimize implementations for certain application scenarios. Moreover, the work presented in Chapter 6 shows that usage of better suited NTT algorithms over the standard Cooley-Tukey decimation-in-time algorithm yields certain improvement. For future work it seems beneficial to review previous work on the implementation of the fast Fourier transform (FFT) which could also be used in the NTT case. Moreover, the Fermat [BS06] and Mersenne variants of the NTT [Nus82] could lead to better performance for certain parameter sets. Another field of further research could be the examination and comparison of the NTT to multiplication algorithms like Nussbaumer multiplication [Nus80, Nus82], which was applied in [BCNS15]. With regard to Gaussian sampling, many algorithms have been examined in this thesis. However, a Gaussian sampler is still an expensive part in most implementations, so that further optimizations or research on different algorithms seem worthwhile.

Investigation of Alternative Schemes

Currently, the performance of RLWEenc, GLP, and BLISS is quite well understood. However, different schemes might turn out to be more efficient, more secure, or simpler to implement. Examples of signature schemes that might perform well on embedded systems are BG [BG14], PASSSign [HPS+14], the modified NTRU signature scheme [MBDG14], or the signature scheme from [DLP14]. Another interesting application of ideal lattice-based cryptography is (authenticated) key exchange. Key exchange protocols can either be realized using public-key encryption and signatures (like RLWEenc and BLISS) or with specialized protocols like [FSXY12, BCNS15, ZZD+15] and it is not yet clear which approach is more efficient and secure. It would also be an interesting area of future research to apply some recent results for ideal lattice-based schemes to NTRU (as in [MBDG14]) and to revisit previous works regarding the implementation of NTRU. Whether NTRU-based or ideal lattice-based schemes will lead to better post-quantum cryptosystems does not seem to be finally settled, yet.

Implementation of Advanced Schemes

Due to the flexibility of lattice-assumptions more complex or advanced schemes than just public-key encryption or signature schemes can be constructed. One example are IBE schemes, which promise simpler key management [Sha84]. Judged by the required computations and proposed parameter sets it seems that relatively simple lattice-based IBE schemes like the one presented in [DLP14] could also be realized efficiently on constrained devices. Another interesting field could be research on ABE [Boy13]. Regarding homomorphic encryption only few works looked at techniques for acceleration in hardware and by tweaking the schemes or implementations it should be possible to significantly improve on the current results. Additionally, the implementation of FPGA-based accelerators for schemes like LTV [LTV12] or BGV [BGV12] instead of YASHE appears to be possible future work.

Bibliography

[AB74] Ramesh C. Agarwal and Charles Sidney Burrus. Fast convolution using Fermat number transforms with applications to digital filtering. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 22(2):87 – 97, April 1974. [Cited on page 48.]

[ABF$^+$08] Ali Can Atici, Lejla Batina, Junfeng Fan, Ingrid Verbauwhede, and Siddika Berna Örs. Low-cost implementations of NTRU for pervasive security. In *19th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2008, July 2-4, 2008, Leuven, Belgium*, pages 79–84. IEEE Computer Society, 2008. [Cited on pages 28, 63, and 78.]

[ACF$^+$15] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Des. Codes Cryptography*, 74(2):325–354, 2015. [Cited on page 31.]

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, 2009. [Cited on page 13.]

[AFG13] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013. [Cited on page 31.]

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011. [Cited on page 34.]

[AH08] Bijan Ansari and M. Anwar Hasan. High-performance architecture of elliptic curve scalar multiplication. *IEEE Trans. Computers*, 57(11):1443–1453, 2008. [Cited on page 116.]

[AHMN13] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. *J. Cryptology*, 26(2):313–339, 2013. [Cited on pages 97 and 98.]

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. [Cited on page 117.]

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 99–108. ACM, 1996. [Cited on page 12.]

[Ajt98] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 10–19. ACM, 1998. [Cited on page 11.]

[AKS01] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 601–610. ACM, 2001. [Cited on page 11.]

[APS13] Aydin Aysu, Cameron Patterson, and Patrick Schaumont. Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 81–86. IEEE Computer Society, 2013. [Cited on pages 4, 46, 47, 58, 59, 60, 61, 82, 143, 154, and 195.]

[AS15] Aydin Aysu and Patrick Schaumont. Precomputation methods for faster and greener post-quantum cryptography on emerging embedded platforms. *IACR Cryptology ePrint Archive*, 2015:288, 2015. [Cited on page 4.]

[AYK00] Murat Aydos, Tugrul Yanik, and Çetin Kaya Koç. An high-speed ECC-based wireless authentication protocol on an ARM microprocessor. In *16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA*, pages 401–410. IEEE Computer Society, 2000. [Cited on page 126.]

[Baa99] Bevan M. Baas. *An Approach to Low-Power, High Performance, Fast Fourier Transform Processor Design.* PhD thesis, Stanford University, Stanford, CA, USA, 1999. [Cited on pages 19, 132, 133, 135, and 137.]

[Baa05] Bevan M. Baas. A generalized cached-FFT algorithm. In *2005 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP '05, Philadelphia, Pennsylvania, USA, March 18-23, 2005*, pages 89–92. IEEE, 2005. [Cited on pages 19, 132, 133, 134, 135, 137, and 193.]

[Bab86] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986. [Cited on page 31.]

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986. [Cited on pages 49, 50, 87, and 144.]

[BB13] Rachid El Bansarkhani and Johannes A. Buchmann. Improvement and efficient implementation of a lattice-based signature scheme. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 48–67. Springer, 2013. [Cited on pages 96, 121, and 122.]

[BBD08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post Quantum Cryptography.* Springer-Verlag Berlin Heidelberg, 1st edition, 2008. [Cited on pages 2, 7, 29, and 30.]

[BBD$^+$14] Christian H. Bischof, Johannes A. Buchmann, Özgür Dagdelen, Robert Fitzpatrick, Florian Göpfert, and Artur Mariano. Nearest planes in practice. In Berna Ors and Bart Preneel, editors, *Cryptography and Information Security in the Balkans - First International Conference, BalkanCryptSec 2014, Istanbul, Turkey, October 16-17, 2014, Revised Selected Papers*, volume 9024 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2014. [Cited on page 31.]

[BBL$^+$14] Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. SPRING: fast pseudorandom functions from rounded ring products. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2014. [Cited on page 58.]

[BCE$^+$01] Daniel V. Bailey, Daniel Coffin, Adam J. Elbirt, Joseph H. Silverman, and Adam D. Woodbury. NTRU in constrained devices. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer, 2001. [Cited on page 78.]

[BCG$^+$12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012. [Cited on page 149.]

[BCG+13] Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete Ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2013. [Cited on pages 21, 96, 97, 111, and 122.]

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015. [Cited on pages 33, 64, 80, 97, 150, 154, and 155.]

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011. [Cited on pages 115 and 122.]

[BDP+12] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview, 2012. Version 3.2, see `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`. [Cited on page 128.]

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013. [Cited on pages 97 and 125.]

[BEE+12a] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012. [Cited on pages 3, 129, and 202.]

[BEE+12b] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. *IACR Cryptology ePrint Archive*, 2012:507, 2012. [Cited on page 203.]

[Ber69] G. Bergland. Fast Fourier transform hardware implementations–an overview. *Audio and Electroacoustics, IEEE Transactions on*, 17(2):104 – 108, June 1969. [Cited on pages 46, 47, and 61.]

[Ber14] Daniel J. Bernstein. A subfield-logarithm attack against ideal lattices, 2014. See `http://blog.cr.yp.to/20140213-ideal.html` (accessed 2015-06-04). [Cited on page 31.]

[BERW08] Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, and Christopher Wolf. Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2008. [Cited on page 115.]

[BFKL93] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 1993. [Cited on page 12.]

[BG09] Céline Blondeau and Benoît Gérard. On the data complexity of statistical attacks against block ciphers (full version). *IACR Cryptology ePrint Archive*, 2009:64, 2009. [Cited on page 23.]

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2014. [Cited on pages 96, 118, and 155.]

[BGJT14] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014. [Cited on pages 1 and 29.]

[BGL$^+$14] Hai Brenner, Lubos Gaspar, Gaëtan Leurent, Alon Rosen, and François-Xavier Standaert. FPGA implementations of SPRING - and their countermeasures against side-channel attacks. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 414–432. Springer, 2014. [Cited on page 58.]

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *IACR Cryptology ePrint Archive*, 2011:277, 2011. [Cited on page 149.]

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012. [Cited on pages 43, 131, 132, 148, 149, and 155.]

[BJ14] Ahmad Boorghany and Rasool Jalili. Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers. *IACR Cryptology ePrint Archive*, 2014:78, 2014. [Cited on pages 81, 82, 88, 93, 94, 96, 118, 129, and 130.]

[BKPS07] Selçuk Baktir, Sandeep Kumar, Christof Paar, and Berk Sunar. A state-of-the-art elliptic curve cryptographic processor operating in the frequency domain. *Mob. Netw. Appl.*, 12(4):259–270, August 2007. [Cited on page 47.]

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003. [Cited on page 31.]

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to` (accessed 2013-05-10). [Cited on page 78.]

[Bla10] Richard E. Blahut. *Fast Algorithms for Signal Processing*. Cambridge University Press, 2010. [Cited on pages 8, 15, and 48.]

[BLLN13a] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2013:75, 2013. [Cited on page 67.]

[BLLN13b] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013. [Cited on pages 9, 40, 42, 43, and 132.]

[BLN14] Joppe W. Bos, Kristin E. Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics*, 50:234–243, 2014. [Cited on page 131.]

[BLP$^+$13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 575–584. ACM, 2013. [Cited on pages 13, 30, 64, 65, 72, 73, 77, and 91.]

[BMV06] Johannes Buchmann, Alexander May, and Ulrich Vollmer. Perspectives for cryptographic long-term security. *Commun. ACM*, 49(9):50–55, 2006. [Cited on page 29.]

[BNP+11] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 389–400. ACM, 2011. [Cited on pages 4 and 203.]

[Boy13] Xavier Boyen. Attribute-based functional encryption on lattices. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 122–142. Springer, 2013. [Cited on page 155.]

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012. [Cited on pages 67 and 131.]

[BS06] Selçuk Baktir and Berk Sunar. Achieving efficient polynomial multiplication in Fermat fields using the fast Fourier transform. In Ronaldo Menezes, editor, *Proceedings of the 44st Annual Southeast Regional Conference, 2006, Melbourne, Florida, USA, March 10-12, 2006*, pages 549–554. ACM, 2006. [Cited on pages 46, 48, and 154.]

[BSJ14] Ahmad Boorghany, Siavash Bayat Sarmadi, and Rasool Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *IACR Cryptology ePrint Archive*, 2014:514, 2014. [Cited on pages 81, 82, 83, 84, 86, 87, 88, 93, 94, 96, 118, 129, and 130.]

[BSTV04] Jean-Luc Beuchat, Nicolas Sendrier, Arnaud Tisserand, and Gilles Villard. FPGA implementation of a recently published signature scheme. *[Research Report] RR-5158, INRIA.*, 2004. <inria-00077045>, See https://hal.inria.fr/inria-00077045/document. [Cited on page 115.]

[CA74] Hui-Chuan Chen and Yoshinori Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974. [Cited on page 23.]

[Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006. [Cited on pages 97, 101, and 104.]

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris

Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009. [Cited on page 122.]

[CFA06] Henri Cohen, Gerhard Frey, and Roberto M. Avanzi. *Handbook of elliptic and hyperelliptic curve cryptography.* Chapman & Hall/CRC, 2006. [Cited on page 50.]

[CG00] Eleanor Chu and Alan George. *Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms.* CRC Press, Boca Raton, FL, USA, 2000. [Cited on pages 19, 82, 83, 84, and 133.]

[Cha12] Kenneth Chang. I.B.M. researchers inch toward quantum computer. New York Times Article, February 28, 2012. See http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?_r=1&hpw. [Cited on pages 1 and 29.]

[CKK15] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Search-and-compute on encrypted data. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2015. [Cited on page 131.]

[CKL15] Jung Hee Cheon, Miran Kim, and Kristin E. Lauter. Homomorphic computation of edit distance. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2015. [Cited on page 131.]

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition edition, 7 2009. [Cited on pages 18 and 197.]

[CMNT11] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 487–504. Springer, 2011. [Cited on pages 131 and 148.]

[CMO$^+$13] Xiaolin Cao, Ciara Moore, Máire O'Neill, Elizabeth O'Sullivan, and Neil Hanley. Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction. *IACR Cryptology ePrint Archive*, 2013:616, 2013. conference version of [CMO$^+$14]. [Cited on page 165.]

[CMO$^+$14] Xiaolin Cao, Ciara Moore, Máire O'Neill, Neil Hanley, and Elizabeth O'Sullivan. High-speed fully homomorphic encryption over the integers. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers*, volume 8438 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2014. extended version: [CMO+13]. [Cited on pages 132, 148, and 164.]

[CMV+14] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-speed polynomial multiplication architecture for Ring-LWE and SHE cryptosystems. *IACR Cryptology ePrint Archive*, 2014:646, 2014. [Cited on pages 4, 46, 47, 58, 59, 60, 61, 132, and 195.]

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011. [Cited on pages 31 and 38.]

[Com90] Paul G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990. [Cited on page 15.]

[CP01] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2001. [Cited on pages 19 and 82.]

[CRPS12] David Cousins, Kurt Rohloff, Chris Peikert, and Richard E. Schantz. An update on SIPHER (scalable implementation of primitives for homomorphic encryption) - FPGA implementation using simulink. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pages 1–5. IEEE, 2012. [Cited on page 148.]

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965. [Cited on pages 17, 83, and 133.]

[CT91] Thomas M. Cover and Joy Thomas. *Elements of Information Theory*. Wiley, 1991. [Cited on page 23.]

[CWB14] Daniel Cabarcas, Patrick Weiden, and Johannes Buchmann. On the efficiency of provably secure NTRU. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2014. [Cited on page 63.]

[DB13] Léo Ducas-Binda. *Signatures Fondées sur les Réseaux Euclidiens: Attaques, Analyses et Optimisations*. PhD thesis, École Normale Supérieure Paris, 2013. `http://www.di.ens.fr/~ducas/Thesis/thesis.pdf`. [Cited on pages 8, 12, 13, 14, and 20.]

[DBG+14] Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. High-speed signatures from standard lattices. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers*, volume 8895 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2014. [Cited on pages 3, 96, 117, 118, 119, 122, and 201.]

[dCRVV15] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 339–344. ACM, 2015. [Cited on pages 4, 82, 83, 84, 88, 93, 118, and 130.]

[dCUHV13] Ruan de Clercq, Leif Uhsadel, Anthony Van Herrewege, and Ingrid Verbauwhede. Ultra low-power implementation of ECC on the ARM cortex-M0+. *IACR Cryptology ePrint Archive*, 2013:609, 2013. [Cited on page 127.]

[DDLL13a] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013. Full version available at `http://eprint.iacr.org/2013/383.pdf`. [Cited on pages 20, 21, 22, 38, 40, 41, 45, 73, 75, 95, 96, 97, 103, 105, 111, 118, 122, 127, 166, and 195.]

[DDLL13b] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. *IACR Cryptology ePrint Archive*, 2013:383, 2013. Full version of [DDLL13a]. [Cited on pages 40, 65, 69, and 128.]

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986. See `http://luc.devroye.org/rnbookindex.html`. [Cited on pages 23, 69, 88, and 97.]

[DG07] Markus Dichtl and Jovan Dj. Golic. High-speed true random number generation with logic gates only. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 45–62. Springer, 2007. [Cited on page 97.]

[DG14] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng. Commun. Comput.*, 25(3):159–180, 2014. [Cited on pages 8, 19, 20, 21, 33, 64, 69, 71, 88, 96, 97, 111, 115, 122, and 123.]

[DGK+12] Benedikt Driessen, Tim Güneysu, Elif Bilge Kavun, Oliver Mischke, Christof Paar, and Thomas Pöppelmann. IPSecco: A lightweight and reconfigurable IPSec core. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012, Cancun, Mexico, December 5-7, 2012*, pages 1–7. IEEE, 2012. [Cited on pages 3 and 202.]

[DHH+15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514, 2015. [Cited on page 94.]

[DHS14] Yarkin Doröz, Yin Hu, and Berk Sunar. Homomorphic AES evaluation using NTRU. *IACR Cryptology ePrint Archive*, 2014:39, 2014. [Cited on page 149.]

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2014. [Cited on pages 64, 88, 96, and 155.]

[DMH+11] Michael Dreschmann, Joachim Meyer, Michael Hübner, Rene Schmogrow, David Hillerkuss, Jürgen Becker, Juerg Leuthold, and Wolfgang Freude. Implementation of an ultra-high speed 256-point FFT for Xilinx Virtex-6 devices. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 829 –834, july 2011. [Cited on page 47.]

[DN12] Léo Ducas and Phong Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2012. [Cited on pages 28 and 95.]

[DÖS13] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. Evaluating the hardware performance of a million-bit multiplier. In *2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013*, pages 955–962. IEEE, 2013. [Cited on pages 132 and 148.]

[DÖS15] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Trans. Computers*, 64(6):1509–1521, 2015. [Cited on pages 132, 137, and 149.]

[DS07] J.P. Deschamps and G. Sutter. Comparison of FPGA implementation of the mod M reduction. *Latin American applied research*, 37(1):93–97, 2007. [Cited on page 49.]

[Duc14] Léo Ducas. Accelerating Bliss: the geometry of ternary polynomials. *IACR Cryptology ePrint Archive*, 2014:874, 2014. [Cited on page 115.]

[EDW+14] Maik Ender, Gerd Düppmann, Alexander Wild, Thomas Pöppelmann, and Tim Güneysu. A hardware-assisted proof-of-concept for secure VoIP clients on untrusted operating systems. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig14, Cancun, Mexico, December 8-10, 2014*, pages 1–6. IEEE, 2014. [Cited on pages 3 and 201.]

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. MicroEliece: McEliece for embedded devices. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2009. [Cited on page 115.]

[EHL14] Kirsten Eisenträger, Sean Hallgren, and Kristin E. Lauter. Weak instances of PLWE. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 183–194. Springer, 2014. [Cited on page 73.]

[EHvM$^+$10] Thomas Eisenbarth, Stefan Heyse, Ingo von Maurich, Thomas Poeppelmann, Johannes Rave, Cornel Reuber, and Alexander Wild. Evaluation of SHA-3 candidates for 8-bit embedded processors. *The Second SHA-3 Candidate Conference, Santa Barbara, California, USA, 2010*, 2010. See `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/HEYSE_EvaluationSHA-3Candidatesfor8-bitProcessors.pdf`. [Cited on page 4.]

[ELOS15] Yara Elias, Kristin E. Lauter, Ekin Ozman, and Katherine E. Stange. Provably weak instances of Ring-LWE. *IACR Cryptology ePrint Archive*, 2015:106, 2015. [Cited on pages 31 and 73.]

[Eme09] Pavel Emeliyanenko. Efficient multiplication of polynomials on graphics hardware. In Yong Dou, Ralf Gruber, and Josef M. Joller, editors, *Advanced Parallel Processing Technologies, 8th International Symposium, APPT 2009, Rapperswil, Switzerland, August 24-25, 2009, Proceedings*, volume 5737 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2009. [Cited on page 47.]

[EW11] Niall Emmart and Charles C. Weems. High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. *Parallel Processing Letters*, 21(3):359–375, 2011. [Cited on page 148.]

[Fit14] Robert Fitzpatrick. *Some Algorithms for Learning with Errors*. PhD thesis, Royal Holloway and Bedford New College, University of London, August 2014. See `https://pure.royalholloway.ac.uk/portal/files/22811572/Thesis.pdf`. [Cited on page 31.]

[Fol14] Janos Follath. Gaussian sampling in lattice based cryptography. *Tatra Mountains Mathematical Publications*, 60(3):1–23, 2014. [Cited on pages 8 and 20.]

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986. [Cited on pages 38 and 96.]

[FSXY12] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes A. Buchmann, and Mark Manulis, editors, *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, volume 7293 of *Lecture Notes in Computer Science*, pages 467–484. Springer, 2012. [Cited on page 155.]

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012. [Cited on page 43.]

[Gal12] Steven D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, Cambridge, New York, 2012. [Cited on page 28.]

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (IoT): A vision, architectural elements, and future directions. *Future Generation Comp. Syst.*, 29(7):1645–1660, 2013. [Cited on page 117.]

[GCB13] Tamas Györfi, Octavian Cret, and Zalan Borsos. Implementing modular FFTs in FPGAs - A basic block for lattice-based cryptography. In *2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013*, pages 305–308. IEEE, 2013. [Cited on page 58.]

[GCHB12] Tamas Györfi, Octavian Cret, Guillaume Hanrot, and Nicolas Brisebarre. High-throughput hardware architecture for the SWIFFT / SWIFFTX hash functions. *IACR Cryptology ePrint Archive*, 2012:343, 2012. [Cited on pages 46 and 58.]

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009. [Cited on page 131.]

[GFS$^+$12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes A. Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2012. [Cited on pages 2, 33, 34, 46, 57, 58, 60, 61, 64, 65, 66, 67, 69, 77, 78, and 79.]

[GG03] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003. [Cited on pages 8 and 15.]

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 1997. [Cited on pages 2 and 95.]

[GH11] Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011. [Cited on pages 131 and 148.]

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012. [Cited on page 132.]

[GKA$^+$10] Kris Gaj, Jens-Peter Kaps, Venkata Amirineni, Marcin Rogawski, Ekawat Homsirikamol, and Benjamin Y. Brewster. ATHENa - Automated Tool for Hardware EvaluatioN: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs. In *International Conference on Field Programmable Logic and Applications, FPL 2010, August 31 2010 - September 2, 2010, Milano, Italy*, pages 414–421. IEEE, 2010. [Cited on page 55.]

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, 2012. [Cited on pages 2, 3, 34, 36, 38, 45, 46, 59, 95, 96, 97, 106, 113, 114, 116, 118, 195, and 202.]

[GLP15] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Lattice-based signatures: Optimization and implementation on reconfigurable hardware. *IEEE Trans. Computers*, 64(7):1954–1967, 2015. [Cited on pages 3, 7, 27, 34, 36, 38, 45, 61, 95, 195, and 201.]

[GN08a] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within Mordell's inequality. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 207–216. ACM, 2008. [Cited on page 11.]

[GN08b] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008. [Cited on page 38.]

[GØJ$^+$11] Danilo Gligoroski, Rune Steinsmo Ødegård, Rune Erlend Jensen, Ludovic Perret, Jean-Charles Faugère, Svein Johan Knapskog, and Smile Markovski. MQQ-SIG - an ultra-fast and provably CMA resistant digital signature scheme. In Liqun

Chen, Moti Yung, and Liehuang Zhu, editors, *Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China, November 27-29, 2011, Revised Selected Papers*, volume 7222 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2011. [Cited on page 122.]

[Gol06] Jovan Dj. Golic. New methods for digital generation and postprocessing of random data. *IEEE Trans. Computers*, 55(10):1217–1229, 2006. [Cited on page 97.]

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2013. [Cited on pages 3, 82, 96, 101, 117, 118, 119, 121, 122, and 202.]

[GP08] Tim Güneysu and Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2008. [Cited on pages 47, 78, 79, 114, and 116.]

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 197–206. ACM, 2008. [Cited on pages 12, 19, 69, 96, 97, and 121.]

[GPW$^+$04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2004. [Cited on pages 15, 81, 90, 93, 94, and 129.]

[GRH11] Vinay Gautam, Kailash Chandra Ray, and Pauline Haddow. Hardware efficient design of variable length FFT processor. In Rolf Kraemer, Adam Pawlak, Andreas Steininger, Mario Schölzel, Jaan Raik, and Heinrich Theodor Vierhaus, editors, *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2011, Cottbus, Germany, April 13-15, 2011*, pages 309–312. IEEE, 2011. [Cited on pages 46 and 47.]

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996. [Cited on pages 29 and 38.]

[GS66] W. Morven Gentleman and G. Sande. Fast Fourier transforms: for fun and profit. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, volume 29 of *AFIPS Conference Proceedings*, pages 563–578. AFIPS / ACM / Spartan Books, Washington D.C., 1966. [Cited on page 83.]

[GSS$^+$11] Benjamin Glas, Oliver Sander, Vitali Stuckert, Klaus D. Müller-Glaser, and Jürgen Becker. Prime field ECDSA signature processing for reconfigurable embedded systems. *Int. J. Reconfig. Comp.*, 2011:836460:1–836460:12, 2011. [Cited on pages 114 and 116.]

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013. [Cited on page 131.]

[GTV12] Roberto Gutierrez, V. Torres, and Javier Valls. Hardware architecture of a Gaussian noise generator based on the inversion method. *IEEE Trans. on Circuits and Systems*, 59-II(8):501–505, 2012. [Cited on pages 96 and 97.]

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *J. Symb. Comput.*, 60:113–119, 2014. [Cited on page 82.]

[HG12] Stefan Heyse and Tim Güneysu. Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2012. [Cited on pages 78 and 79.]

[HHHW09] Philip S. Hirschhorn, Jeffrey Hoffstein, Nick Howgrave-Graham, and William Whyte. Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, volume 5536 of *Lecture Notes in Computer Science*, pages 437–455, 2009. [Cited on pages 63, 78, and 93.]

[HHP$^+$03] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: digital signatures using the NTRU lattice. In Marc Joye, editor, *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, volume 2612 of *Lecture Notes in Computer Science*, pages 122–140. Springer, 2003. [Cited on pages 28, 95, and 115.]

[HPO+15] James Howe, Thomas Pöppelmann, Máire O'Neill, Elizabeth O'Sullivan, and Tim Güneysu. Practical lattice-based digital signature schemes. *ACM Trans. Embedded Comput. Syst.*, 14(3):41, 2015. [Cited on pages 3, 7, 27, 96, 117, and 201.]

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998. [Cited on pages 2, 28, 40, and 63.]

[HPS08] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An introduction to mathematical cryptography.* Springer Verlag, 2008. [Cited on pages 11 and 28.]

[HPS11] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, volume 6639 of *Lecture Notes in Computer Science*, pages 159–190. Springer, 2011. [Cited on pages 11 and 13.]

[HPS+14] Jeffrey Hoffstein, Jill Pipher, John M. Schanck, Joseph H. Silverman, and William Whyte. Practical signatures from the partial Fourier recovery problem. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, volume 8479 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2014. [Cited on pages 96, 115, 118, 122, and 155.]

[HS13] Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR microcontrollers. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013. [Cited on pages 81, 129, and 130.]

[HS14] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014. Source code and documentation: `https://shaih.github.io/HElib/`. [Cited on page 149.]

[HS15] Michael Hutter and Peter Schwabe. Multiprecision multiplication on AVR revisited. *J. Cryptographic Engineering*, 5(3):201–214, 2015. [Cited on page 15.]

[HvMG13] Stefan Heyse, Ingo von Maurich, and Tim Güneysu. Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and*

Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings, volume 8086 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2013. [Cited on pages 93 and 94.]

[HVP10] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed records for NTRU. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco, CA, USA, March 1-5, 2010. Proceedings*, volume 5985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2010. [Cited on pages 28, 63, and 78.]

[HW11] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011. [Cited on page 90.]

[JA11] Bernhard Jungk and Jürgen Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In Peter M. Athanas, Jürgen Becker, and René Cumplido, editors, *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011*, pages 235–241. IEEE Computer Society, 2011. [Cited on page 107.]

[JF11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. [Cited on page 29.]

[Jou13] Antoine Joux. A new index calculus algorithm with complexity $l(1/4+o(1))$ in small characteristic. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 355–379. Springer, 2013. [Cited on pages 1 and 29.]

[JS07] Team Member Kimmo Järvinen and Jorma Skyttä. Final project report: Cryptoprocessor for elliptic curve digital signature algorithm (ECDSA), 2007. See `http://www.altera.com/literature/dc/2007/in_2007_dig_signature.pdf`. [Cited on pages 114 and 116.]

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 193–206. ACM, 1983. [Cited on page 11.]

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 1951. [Cited on page 23.]

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. [Cited on pages 28 and 29.]

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. [Cited on page 123.]

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963. [Cited on page 91.]

[Koe13] John Koetsier. An inside look at the world's newest quantum computing and nanotechnology center, 2013. See `http://venturebeat.com/2013/05/15/an-inside-look-at-the-worlds-newest-quantum-computing-and-nanotechnology-center/`. [Cited on pages 1 and 29.]

[KY09] A.A. Kamal and A.M. Youssef. An FPGA implementation of the NTRUEncrypt cryptosystem. In *Microelectronics (ICM), 2009 International Conference on, Marrakech, Morocco, December 19-22, 2009*, pages 209–212. IEEE, 2009. [Cited on pages 28, 63, 78, and 79.]

[Lan14] Adeline Langlois. *Lattice-Based Cryptography: Security Foundations and Constructions*. PhD thesis, ENS de Lyon, October 2014. `https://tel.archives-ouvertes.fr/tel-01126931`. [Cited on page 8.]

[Lep14] Tancrède Lepoint. *Design and Implementation of Lattice-Based Cryptography*. PhD thesis, École Normale Supérieure and University of Luxembourg, June 2014. See `https://tel.archives-ouvertes.fr/tel-01069864`. [Cited on pages 8 and 20.]

[LGK10] Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010), Tokyo, Japan, November 29, 2010*. IEEE Computer Society Press, 2010. [Cited on pages 93, 94, and 129.]

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982. [Cited on pages 11 and 30.]

[LM06] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2006. [Cited on page 95.]

[LM09] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology*

Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings, volume 5677 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2009. [Cited on page 13.]

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2008. [Cited on pages 17, 50, 58, and 95.]

[LN13] Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In Ed Dawson, editor, *Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013, San Francisco,CA, USA, February 25-March 1, 2013. Proceedings*, volume 7779 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2013. [Cited on pages 31, 33, and 64.]

[LN14] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2014. [Cited on pages 42, 43, 132, 149, 150, and 196.]

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011. [Cited on pages 19, 27, 31, 32, 33, 34, 45, 64, 67, and 82.]

[LPR$^+$10a] Hans Löhr, Thomas Pöppelmann, Johannes Rave, Martin Steegmanns, and Marcel Winandy. Trusted virtual domains on OpenSolaris: Usable secure desktop environments. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing, STC 2010, Chicago, IL, USA, October 4, 2010*, pages 91–96. ACM, 2010. [Cited on pages 4 and 203.]

[LPR10b] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010. Presentation slides: `http://crypto.rd.francetelecom.com/events/eurocrypt2010/talks/slides-ideal-lwe.pdf`. [Cited on pages 2, 14, 27, 31, 40, 45, 46, 56, 64, 82, 95, 176, and 177.]

[LPR10c] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings, 2010. Presentation of [LPR10b] given by Chris Peikert at

Eurocrypt'10. See `http://www.cc.gatech.edu/~cpeikert/pubs/slides-ideal-lwe.pdf`. [Cited on pages 31, 64, and 82.]

[LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *IACR Cryptology ePrint Archive*, 2012:230, 2012. Full version of [LPR10b]. [Cited on page 31.]

[LS12] Gregory Landais and Nicolas Sendrier. Implementing CFS. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, volume 7668 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2012. [Cited on page 122.]

[LSR$^+$15] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient Ring-LWE encryption on 8-bit AVR processors. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 663–682. Springer, 2015. [Cited on pages 87 and 94.]

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1219–1234. ACM, 2012. [Cited on pages 40, 131, 132, 149, and 155.]

[Lyu08a] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In Ronald Cramer, editor, *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, volume 4939 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 2008. [Cited on page 96.]

[Lyu08b] Vadim Lyubashevsky. *Towards Practical Lattice-Based Cryptography*. PhD thesis, University of California, San Diego, 2008. `http://www.di.ens.fr/~lyubash/papers/dissertation_singlespaced.pdf`. [Cited on pages 8 and 9.]

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009. [Cited on pages 34, 95, and 96.]

[Lyu11] Vadim Lyubashevsky. Lattice signatures without trapdoors. *IACR Cryptology ePrint Archive*, 2011:537, 2011. [Cited on page 20.]

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 -*

31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012. [Cited on pages 34, 46, 70, 96, and 121.]

[MBDG14] Carlos Aguilar Melchor, Xavier Boyen, Jean-Christophe Deneuville, and Philippe Gaborit. Sealing the leak on classical NTRU signatures. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2014. [Cited on pages 96, 115, 121, and 155.]

[MBFK14] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIRe: Private information retrieval for everyone. *IACR Cryptology ePrint Archive*, 2014:1025, 2014. [Cited on pages 4, 82, and 118.]

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2002. [Cited on pages 7 and 11.]

[Mic07] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007. [Cited on page 95.]

[Mic11] Daniele Micciancio. The geometry of lattice cryptography. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volume 6858 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 2011. [Cited on pages 10 and 11.]

[Mic14] Daniele Micciancio. CSE206A: Lattices algorithms and applications (spring 2014), 2014. Lecture notes of a course given in UCSD. See `http://cseweb.ucsd.edu/classes/sp14/cse206A-a/`. [Cited on pages 8 and 11.]

[MLPJ13] Yuan Ma, Zongbin Liu, Wuqiong Pan, and Jiwu Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 421–437. Springer, 2013. [Cited on page 116.]

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 1985. [Cited on pages 49 and 144.]

[Mon08] Mariano Monteverde. NTRU software implementation for constrained devices. Master's thesis, Katholieke Universiteit Leuven, 2008. See `https://www.cosic.esat.kuleuven.be/publications/thesis-161.pdf`. [Cited on page 93.]

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007. [Cited on page 72.]

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718. Springer, 2012. [Cited on pages 96 and 121.]

[MP13] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2013. [Cited on pages 13, 25, and 34.]

[MR04] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measures. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 372–381. IEEE Computer Society, 2004. [Cited on pages 12 and 179.]

[MR07] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007. Full version of [MR04]. [Cited on page 12.]

[MR09] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Chapter in Post-quantum Cryptography*, pages 147–191. Springer, 2009. [Cited on pages 7, 10, 11, and 12.]

[MS06] Florence MacWilliams and Neil Sloane. *The theory of error-correcting codes*. North-Holland, 2006. [Cited on page 68.]

[NLV11] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 113–124. ACM, 2011. [Cited on pages 46, 56, 57, 59, and 131.]

[NR09] Phong Q. Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. *J. Cryptology*, 22(2):139–160, 2009. [Cited on pages 28, 95, and 115.]

[Nus80] Henri J. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(2):205–215, Apr 1980. [Cited on pages 94 and 154.]

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*, volume 2 of *Springer Series in Information Sciences*. Springer, Berlin, DE, 1982. [Cited on pages 8, 15, 16, 19, 94, 150, and 154.]

[NV09] Phong Q. Nguyen and Brigitte Valle. *The LLL Algorithm: Survey and Applications.* Springer-Verlag Berlin Heidelberg, 1st edition, 2009. [Cited on page 11.]

[Ode13] Tobias Oder. Efficient microcontroller implementation of the bimodal lattice signature scheme, November 2013. Bachelor's thesis, Hardware Security Group, Ruhr-University Bochum. Supervised by Prof. Dr.-Ing. Tim Güneysu and Dipl.-Ing. Thomas Pöppelmann. [Cited on page 117.]

[OPG14] Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 1–6. ACM, 2014. [Cited on pages 3, 7, 27, 96, 117, and 202.]

[PCC$^+$14] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 13–24. IEEE Computer Society, 2014. [Cited on pages 132, 136, 147, and 149.]

[PDG14a] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2014. [Cited on pages 3, 7, 27, 61, 82, 83, 84, 95, 103, 104, 106, 116, and 202.]

[PDG14b] Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. *IACR Cryptology ePrint Archive*, 2014:254, 2014. [Cited on pages 3, 7, 22, 24, 27, 95, and 203.]

[Pea68] Marshall C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968. [Cited on pages 46, 47, 52, 59, 61, and 142.]

[Pei09] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 333–342. ACM, 2009. [Cited on pages 12, 13, and 30.]

[Pei10] Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2010. [Cited on pages 21, 22, and 25.]

[Pei14] Chris Peikert. Lattice cryptography for the Internet. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 197–219. Springer, 2014. [Cited on page 64.]

[Per03] Colin Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Math. Comput.*, 72(241):387–395, 2003. [Cited on page 48.]

[PG12] Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2012. [Cited on pages 3, 7, 45, 46, 58, 59, 143, and 202.]

[PG13] Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2013. [Cited on pages 3, 27, 45, 58, 61, 63, 66, 77, 78, 79, and 202.]

[PG14] Thomas Pöppelmann and Tim Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014*, pages 2796–2799. IEEE, 2014. [Cited on pages 3, 63, 91, 92, 111, and 202.]

[PNPM15a] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 143–163. Springer, 2015. [Cited on pages 3, 27, 131, and 201.]

[PNPM15b] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. *IACR Cryptology ePrint Archive*, 2015:631, 2015. [Cited on pages 3, 131, and 203.]

[POG15a] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2015. [Cited on pages 3, 27, 63, 81, 96, 117, and 201.]

[POG15b] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. Speed records for ideal lattice-based cryptography on AVR. *IACR Cryptology ePrint Archive*, 2015:382, 2015. [Cited on pages 3, 81, and 203.]

[Pol71] J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971. [Cited on page 15.]

[Pöp11] Thomas Pöppelmann. Efficient implementation of a digital signature scheme based on low-density compact knapsacks on reconfigurable hardware, November 2011. Diploma thesis (equiv. to Master's thesis), Hardware Security Group, Ruhr-University Bochum. Supervised by Prof. Dr.-Ing. Tim Güneysu. [Cited on page 95.]

[PP09] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag Berlin Heidelberg, 1st edition, 2009. [Cited on page 28.]

[PR06] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 145–166. Springer, 2006. [Cited on page 95.]

[PTBW11] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic public key systems. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 475–490. Springer, 2011. [Cited on page 115.]

[Rad72] C.M. Rader. Discrete convolutions via Mersenne transforms. *IEEE Transactions on Computers*, 100(12):1269–1273, 1972. [Cited on page 48.]

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005. [Cited on pages 2, 12, 13, and 64.]

[Reg09] Oded Regev. Lattices in computer science, 2009. Lecture notes of a course given in Tel Aviv University. See `http://www.cims.nyu.edu/~regev/teaching/lattices_fall_2009/`. [Cited on pages 8, 10, and 11.]

[Reg10] Oded Regev. The learning with errors problem (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, June 9-12, 2010*, pages 191–204. IEEE Computer Society, 2010. [Cited on page 12.]

[RG13] Steven Rich and Barton Gellman. NSA seeks quantum computer that could crack most codes. *The Washington Post*, 2013. See `http://wapo.st/19DycJT`. [Cited on pages 1 and 29.]

[RJV$^+$15] Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil S. Dimitrov, and Ingrid Verbauwhede. Modular hardware architecture for somewhat homomorphic function evaluation. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 164–184. Springer, 2015. [Cited on pages 132 and 149.]

[RRM12] Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay. Pushing the limits of high-speed GF(2^m) elliptic curve. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 494–511. Springer, 2012. [Cited on pages 78 and 79.]

[RRVV14] Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel secure discrete Gaussian sampling. *IACR Cryptology ePrint Archive*, 2014:591, 2014. [Cited on pages 97 and 154.]

[RS10] Markus Rückert and Michael Schneider. Estimating the security of lattice-based cryptosystems. *IACR Cryptology ePrint Archive*, 2010:137, 2010. [Cited on page 59.]

[RVM$^+$14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014. [Cited on pages 4, 18, 46, 47, 58, 59, 60, 61, 64, 66, 67, 77, 78, 79, 82, 83, 84, 107, 135, 137, 143, 146, 154, and 195.]

[RVV13] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. High precision discrete Gaussian sampling on FPGAs. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 383–401. Springer, 2013. [Cited on pages 21, 64, 65, 75, 96, 97, 111, 115, 122, and 123.]

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987. [Cited on page 11.]

[Sch11] Michael Schneider. *Computing Shortest Lattice Vectors on Special Hardware*. PhD thesis, TU Darmstadt, November 2011. See `http://tuprints.ulb.tu-darmstadt.de/2829/`. [Cited on page 8.]

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994. [Cited on page 11.]

[SG14] Pascal Sasdrich and Tim Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels, editors, *ARC*, volume 8405 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2014. [Cited on page 116.]

[Sha84] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, 1984. [Cited on page 155.]

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994. [Cited on pages 1 and 29.]

[SM11] Daisuke Suzuki and Tsutomu Matsumoto. How to maximize the potential of FPGA-based DSPs for modular exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011. [Cited on pages 114 and 116.]

[Soc06] IEEE Vehicular Technology Society. IEEE trial-use standard for wireless access in vehicular environments - security services for applications and management messages. *IEEE Std 1609.2-2006*, pages 0–105, 2006. [Cited on page 1.]

[Sol99] Jerome A. Solinas. Generalized Mersenne numbers, 1999. CACR Technical Report CORR 99-39, Faculty of Mathematics, University of Waterloo. Available from `http://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf`. [Cited on pages 51, 59, and 144.]

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971. [Cited on page 15.]

[SS11] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2011. [Cited on pages 40, 63, and 78.]

[SSHA08] A. Suleiman, H. Saleh, A. Hussein, and D. Akopian. A family of scalable FFT architectures and an implementation of 1024-point radix-2 FFT for real-time communications. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 321 –327, oct. 2008. [Cited on page 47.]

[SSRG11] Rabia Shahid, Malik Umar Sharif, Marcin Rogawski, and Kris Gaj. Use of embedded FPGA resources in implementations of 14 round 2 SHA-3 candidates. In Russell Tessier, editor, *2011 International Conference on Field-Programmable Technology,* *FPT 2011, New Delhi, India, December 12-14, 2011*, pages 1–9. IEEE, 2011. [Cited on page 107.]

[Ste] Richard Stern. Hardware implementations of ECRYPT stream ciphers. VHDL code available from `http://eeweb.poly.edu/faculty/karri/stream_ciphers/trivium.html`, accessed July 25, 2013. [Cited on page 101.]

[STM] STMicroelectronics. UM0586 STM32 Cryptographic Library. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/CD00208802.pdf`. [Cited on pages 126, 127, and 196.]

[Suz07] Daisuke Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2007. [Cited on pages 78, 79, and 97.]

[SWM$^+$10] Abdulhadi Shoufan, Thorsten Wink, H. Gregor Molter, Sorin A. Huss, and Eike Kohnert. A novel cryptoprocessor architecture for the McEliece public-key cryptosystem. *IEEE Trans. Computers*, 59(11):1533–1546, 2010. [Cited on page 115.]

[TLLV07] David B. Thomas, Wayne Luk, Philip Heng Wai Leong, and John D. Villasenor. Gaussian random number generators. *ACM Comput. Surv.*, 39(4), 2007. [Cited on pages 96 and 97.]

[Var08] Michal Varchola. *FPGA Based True Random Number Generators for Embedded Cryptographic Applications*. PhD thesis, Technical University of Kosice, 2008. [Cited on page 70.]

[Vau03] Serge Vaudenay. Decorrelation: A theory for block cipher security. *J. Cryptology*, 16(4):249–286, 2003. [Cited on page 23.]

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. [Cited on page 131.]

[vEB81] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Universiteit van Amsterdam. Mathematisch Instituut, 1981. [Cited on page 11.]

[vMG14] Ingo von Maurich and Tim Güneysu. Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. IEEE, 2014. [Cited on page 79.]

[VOMV96] P.C. Van Oorschot, A.J. Menezes, and S.A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996. [Cited on pages 49, 123, and 144.]

[vzGS05] Joachim von zur Gathen and Jamshid Shokrollahi. Efficient FPGA-based Karatsuba multipliers for polynomials over $\mathbb{F}_2$. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, volume 3897 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2005. [Cited on page 46.]

[WCH14] Wei Wang, Zhilu Chen, and Xinming Huang. Accelerating leveled fully homomorphic encryption using GPU. In *IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014*, pages 2800–2803. IEEE, 2014. [Cited on pages 132 and 148.]

[WH13] Wei Wang and Xinming Huang. FPGA implementation of a large-number multiplier for fully homomorphic encryption. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, May 19-23, 2013*, pages 2589–2592. IEEE, 2013. [Cited on page 148.]

[WHC+12] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using GPU. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pages 1–5. IEEE, 2012. [Cited on page 148.]

[WHC+15] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar. Exploring the feasibility of fully homomorphic encryption. *Computers, IEEE Transactions on*, 64(3):698–706, March 2015. [Cited on pages 132 and 148.]

[WHCB13] Patrick Weiden, Andreas Hülsing, Daniel Cabarcas, and Johannes Buchmann. Instantiating treeless signature schemes. *IACR Cryptology ePrint Archive*, 2013:65, 2013. [Cited on pages 121 and 122.]

[WHEW14] Wei Wang, Xinming Huang, Niall Emmart, and Charles C. Weems. VLSI design of a large-number multiplier for fully homomorphic encryption. *IEEE Trans. VLSI Syst.*, 22(9):1879–1887, 2014. [Cited on pages 132 and 148.]

[Win96] Franz Winkler. *Polynomial Algorithms in Computer Algebra (Texts and Monographs in Symbolic Computation)*. Springer, 1 edition, 8 1996. [Cited on pages 8, 15, 16, and 17.]

[WLT07] Chin-Long Wey, Shin-Yo Lin, and Wei-Chien Tang. Efficient memory-based FFT processors for OFDM applications. In *IEEE International Conference on Electro/Information Technology, 2007*, pages 345 –350, May 2007. [Cited on pages 46 and 47.]

[Xag10] Keita Xagawa. *Cryptography with Lattices*. PhD thesis, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2010. `http://xagawa.net/pdf/2010Thesis.pdf`. [Cited on pages 8 and 11.]

[Xil09a] Xilinx. Spartan-6 FPGA DSP48A1 slice user guide. See `http://www.xilinx.com/support/documentation/user_guides/ug389.pdf`, 2009. [Cited on page 74.]

[Xil09b] Xilinx. XST user guide. See `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/xst.pdf`, 2009. [Cited on page 52.]

[Xil14] Xilinx. Vivado design suite user guide - high-level synthesis. See `http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`, 2014. [Cited on page 50.]

[ZZD⁺15] Jiang Zhang, Zhenfeng Zhang, Jintai Ding, Michael Snook, and Özgür Dagdelen. Authenticated key exchange from ideal lattices. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 719–751. Springer, 2015. [Cited on pages 80 and 155.]

List of Abbreviations

ABE attribute-based encryption

AES advanced encryption standard

AKE authenticated key exchange

ASIC application specific integrated circuit

AVX advanced vector extensions

BG Bai-Galbraith

BGV Brakerski-Gentry-Vaikuntanathan

BKW Blum-Kalai-Wasserman

BLISS bimodal lattice-based signature schemes

CCA chosen ciphertext attack

CDT cumulative distribution table

CPA chosen plaintext attack

CRT Chinese remainder theorem

CT Cooley-Tukey

CVP closest vector problem

DCK decisional compact knapsack

DIT decimation-in-time

DIF decimation-in-frequency

DLP discrete logarithm problem

DRAM dynamic random access memory

DSA digital signature algorithm

DSP digital signal processor

DSPR decisional small polynomial ratio

DSS digital signature scheme

ECC elliptic curve cryptography

ECDSA elliptic curve digital signature algorithm

FF flip-flop

FFT fast Fourier transform

FHE fully homomorphic encryption

FPGA field-programmable gate array

FPU floating point unit

FSM finite-state machine

GLP Güneysu-Lyubashevsky-Pöppelmann

GPU graphics processing unit

GS Gentleman-Sande

HLS high-level synthesis

IBE identity-based encryption

INTT inverse number theoretic transform

IOT Internet of things

KL Kullback-Leibler

LFSR linear feedback shift register

LLL Lenstra-Lenstra-Lovász

LPN learning parity with noise

LTV López-Alt-Tromer-Vaikuntanathan

LWE learning with errors

MAC multiply-accumulate

NAF non-adjacent form

NTRU N-th degree truncated polynomial ring

NTT number theoretic transform

PAR place-and-route

PCIe peripheral component interconnect express

PE processing element

PKE public-key encryption

PPT probabilistic polynomial time

PQC post-quantum cryptography

PRNG pseudo-random number generator

RLWE ring learning with errors

RNG random number generator

ROM read-only memory

RSA Rivest-Shamir-Adleman

RSIS ring short integer solution

SHE somewhat homomorphic encryption

SIMD single instruction multiple data

SIS short integer solution

SIVP shortest independent vectors problem

SMAS2 shifting-addition-multiplication-subtraction-subtraction

SVP shortest vector problem

TLS transport layer security

TRNG true random number generator

YASHE yet another somewhat homomorphic encryption

List of Figures

List of Tables

List of Algorithms

About the Author

Author information as of June 2015.

Personal Data

Name Thomas Pöppelmann

Address Hardware Security Group, ID 2/647, Universitätsstr. 150, 44801 Bochum, Germany

Email thomas.poeppelmann@rub.de

Date of birth August 13, 1986

Place of birth Oelde, Germany

Education

Ruhr University Bochum January 2012 - July 2015

PhD candidate in the Hardware Security Group supervised by Prof. Dr.-Ing. Tim Güneysu and external member of the Research Training Group UbiCrypt. Research with focus on lattice-based cryptography, reconfigurable computing, and embedded systems.

University of Strathclyde, Glasgow September 2010 - January 2011

Semester abroad in the Department of Electronic & Electrical Engineering.

Ruhr University Bochum September 2006 - November 2011

Study of "Security in Information Technology" finished with a diploma (equiv. to a Master's degree). Department of Electrical Engineering and Information Technology.

Professional Experience

Hardware Security Group, Ruhr University Bochum *Bochum, Germany*
Research Assistant January 2012 - June 2015

Involved in German Research Foundation and EU funded projects on post-quantum cryptography. Teaching assistant for bachelor course "Embedded Processors" in 2012 to 2014. Supervision of seminar, bachelor, and master theses. Participation in industry projects.

Microsoft Research *Redmond, USA*
Research Intern September 2014 - December 2014

Hardware implementation of a homomorphic encryption scheme targeting an FPGA-based cloud computing accelerator.

Fraunhofer Institute for Secure Information Technology *Darmstadt, Germany*
Research Intern February 2011 - April 2011

Analysis of vulnerabilities in the usage of the Amazon Web Services (AWS) compute cloud.

LMF, Ruhr University Bochum *Bochum, Germany*
Student Assistant August 2008 - June 2010

Programming of the model test track (scale of 1:2) vehicle control software for a system for fast, reliable, and on-time transportation of goods through underground pipelines in densely populated urban areas (Cargo Cap).

Publications and Academic Activities

In the following, all formal and informal publications of the author of this thesis are listed (information as of June 2015). Additionally, we list invited talks, academic awards, and selected attended conferences, workshops, and summer schools.

Peer-Reviewed Journal Papers

- James Howe, Thomas Pöppelmann, Máire O'Neill, Elizabeth O'Sullivan, and Tim Güneysu. Practical lattice-based digital signature schemes. *ACM Trans. Embedded Comput. Syst.*, 14(3):41, 2015
- Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Lattice-based signatures: Optimization and implementation on reconfigurable hardware. *IEEE Trans. Computers*, 64(7):1954–1967, 2015

Peer-Reviewed Conference Proceeding

- Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 143–163. Springer, 2015
- Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 2015
- Maik Ender, Gerd Düppmann, Alexander Wild, Thomas Pöppelmann, and Tim Güneysu. A hardware-assisted proof-of-concept for secure VoIP clients on untrusted operating systems. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig14, Cancun, Mexico, December 8-10, 2014*, pages 1–6. IEEE, 2014
- Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe. High-speed signatures from standard lattices. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers*, volume 8895 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2014

- Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2014
- Tobias Oder, Thomas Pöppelmann, and Tim Güneysu. Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 1–6. ACM, 2014
- Thomas Pöppelmann and Tim Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014*, pages 2796–2799. IEEE, 2014
- Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 2013
- Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In Philippe Gaborit, editor, *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, volume 7932 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2013
- Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2012
- Benedikt Driessen, Tim Güneysu, Elif Bilge Kavun, Oliver Mischke, Christof Paar, and Thomas Pöppelmann. IPSecco: A lightweight and reconfigurable IPSec core. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012, Cancun, Mexico, December 5-7, 2012*, pages 1–7. IEEE, 2012
- Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2012

- Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, 2012
- Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 389–400. ACM, 2011
- Hans Löhr, Thomas Pöppelmann, Johannes Rave, Martin Steegmanns, and Marcel Winandy. Trusted virtual domains on OpenSolaris: Usable secure desktop environments. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing, STC 2010, Chicago, IL, USA, October 4, 2010*, pages 91–96. ACM, 2010

Workshops without Proceedings

- Thomas Eisenbarth, Stefan Heyse, Ingo von Maurich, Thomas Poeppelmann, Johannes Rave, Cornel Reuber, and Alexander Wild. Evaluation of SHA-3 candidates for 8-bit embedded processors. The Second SHA-3 Candidate Conference, Santa Barbara, California, USA, 2010.

Technical Reports

- Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. Speed records for ideal lattice-based cryptography on AVR. *IACR Cryptology ePrint Archive*, 2015:382, 2015
- Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrián Macías. Accelerating homomorphic evaluation on reconfigurable hardware. *IACR Cryptology ePrint Archive*, 2015:631, 2015
- Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. *IACR Cryptology ePrint Archive*, 2014:254, 2014
- Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. Compact implementation and performance evaluation of hash functions in ATtiny devices. *IACR Cryptology ePrint Archive*, 2012:507, 2012

Invited Talks

- Implementing Lattice-Based Cryptography on Embedded Devices. *Summer school on real-world crypto and privacy 2015, Sibenik, Croatia, June 2015*

- Efficient Implementation of Ideal Lattice-Based Cryptography. *HGI-Kolloquium, Ruhr University Bochum, July 2014*
- Practical Lattice-Based Cryptography. *ClSIT Seminar, Monash University, Melbourne, Australia, June 2014*
- Practical Lattice-Based Signatures. *Aric Seminar, ENS Lyon, France, January 2014*
- Software Speed Records for Lattice-Based Signatures. *CDC Oberseminar, TU Darmstadt, Germany, June 2013*
- Implementation of a Practical Lattice-Based Signature Scheme on Reconfigurable Hardware. *CDC Oberseminar, TU Darmstadt, Germany, June 2012*

Awards and Stipends

- CAST-Förderpreis IT-Sicherheit 2012, first price in category master/diploma thesis
- European Trusted Infrastructure Summer School 2011 (ETISS'11), Darmstadt, Germany, full stipend

Participation in Selected Conferences, Workshops, and Summer Schools

- Summer School on Real-World Crypto and Privacy 2015 (Sibenik, Croatia)
- Real World Crypto Workshop 2015 (London, UK)
- ISCAS 2014 (Melbourne, Australia)
- CryptArchi 2014 (Annecy, France)
- CHES 2013 (Santa Barbara, USA)
- Selected Areas in Cryptography 2013 (Burnaby, Canada)
- CryptArchi 2013 (Fréjus, France)
- PQCrypto 2013 (Limoges, France)
- Keccak & SHA-3 Day 2013 (Brussels, Belgium)
- Crypto for 2020, 2013 (Tenerife, Spain)
- 29th Chaos Communication Congress 2012 (Hamburg, Germany)
- Post-Quantum Cryptography and Quantum Algorithms Workshop 2012 (Leiden, The Netherlands)
- Workshop on Cryptography for the Internet of Things 2012 (Antwerp, Belgium)
- Latincrypt 2012 (Santiago, Chile)
- CHES 2012 (Leuven, Belgium)
- Code-based Cryptography Workshop 2012 (Lyngby, Denmark)
- 2nd Bar-Ilan Winter School on Cryptography: Lattice-Based Cryptography and Applications 2012 (Tel Aviv, Israel)
- European Trusted Infrastructure Summer School 2011 (Darmstadt, Germany)

- Workshop on Cryptography and Security in Clouds 2011 (Zurich, Switzerland)
- Trust 2010 (Berlin, Germany)
- Eurocrypt 2009 (Cologne, Germany)