KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Public Key Cryptography on Hardware Platforms:

Design and Analysis of Elliptic Curve and Lattice-based Cryptoprocessors

Sujoy Sinha Roy

Supervisor:
Prof. dr. ir. I. Verbauwhede
Co-supervisor:
Dr. ir. F. Vercauteren

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

May 2017

Public Key Cryptography on Hardware Platforms:

Design and Analysis of Elliptic Curve and Lattice-based Cryptoprocessors

Sujoy Sinha Roy

Examination committee:
Prof. dr. ir. H. Hens, chair
Prof. dr. ir. P. Verbaeten, deputy chair
Prof. dr. ir. I. Verbauwhede, supervisor
Dr. ir. F. Vercauteren, co-supervisor
Prof. dr. ir. W. Dehaene
Prof. dr. ir. B. Preneel
Prof. dr. D. Mukhopadhyay
(Indian Institute of Technology Kharagpur)
Prof. dr. T. Güneysu
(University of Bremen & DFKI)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

May 2017

Acknowledgements

Swami Vivekananda said, "*Take up one idea. Make that one idea your life–think of it, dream of it, live on that idea. Let the brain, muscles, nerves, every part of your body, be full of that idea, and just leave every other idea alone. This is the way to success.*"

It is not as easy as it seems. In my case, I was lucky enough since prof. Ingrid helped me realize my ideas into reality. I owe this memorable journey to my promotor prof. Ingrid Verbauwhede. Thank you for offering me a PhD position at COSIC and also granting the freedom and flexibility to conduct research. I cannot possibly imagine these five years as a PhD researcher without your guidance and support.

Dr. Frederik Vercauteren, I feel short of words when it comes to you. You played a versatile role in this journey of mine, sometimes a teacher, sometimes a mentor, sometimes a co-author, sometimes a friend indeed, ... and the list continues. I can't thank you enough carefully reviewing my manuscripts and providing me with corrections needed. I am really having a tough time putting your contribution in words, still I tried my level best. I not only found a mentor in you but also a friend to cherish for a lifetime.

I would like to thank my Master thesis supervisor Prof. Debdeep Mukhopadhyay for helping me cultivate my desire to work in the field of cryptography and explore the vast field bit by bit.

I would like to thank my assessors prof. Bart Preneel and prof. Wim Dehaene, and the additional members of the jury prof. Debdeep Mukhopadhyay and prof. Tim Güneysu for their precious time and effort. I would thank prof. Hugo Hens and prof. Pierre Verbaeten for chairing the jury.

I feel lucky to be a part of COSIC and have the opportunity to write papers together with my gifted colleagues Ruan de Clercq, Frederik Vercauteren, Kimmo Järvinen, Oscar Reparaz, Jo Vliegen, Angshuman Karmakar, Zhe Liu, Donald

Chen, Nele Mentens, Junfeng Fan.

Pela, thank you for always helping me out and guiding me in every possible way you could.

Friends, who also happen to be my fellow researchers eased this journey of mine and made it memorable with their brilliant ideas, funny jokes and friendly advice. The company of my fellow researchers always made me forget that I was miles away from home.

Whatever I am today, I owe every bit to my father and mother. They were the light when I felt surrounded by darkness. I cant thank you because no words could ever express what I feel for you. All I have is a small token of thanks for the omnipresent who surrounded me with people who have always brought the best out of me, even when I did not know about my own abilities.

Last but not the least, I would have never got such an opportunity if the European commission wouldn't have shown the encouragement by awarding me the Erasmus scholarship.

A special thanks to my wife Madhu for tolerating my occasional frustrations and encouraging me and thus keeping my heart, mind and soul in working condition at all times.

Abstract

The vast network of connected devices, ranging from tiny Radio Frequency Identification (RFID) tags to powerful desktop computers, generates massive amounts of information. Public-key cryptography (PKC) plays a crucial role in securing this network. In this thesis we focus on the efficient implementation of PKC to address the security challenges of the future.

Aiming to secure resource-constrained connected devices, we design a lightweight elliptic-curve coprocessor for a 283-bit Koblitz curve, which offers 140-bit security. We optimize the scalar conversion which is an important part of point multiplication, and we introduce lightweight countermeasures against timing and power side-channel attacks. The coprocessor consumes only 4.3 KGE.

In the second part of the thesis, we investigate implementation aspects of post-quantum PKC and homomorphic encryption schemes whose security is based on the hardness of the ring-LWE problem. These cryptographic schemes perform arithmetic operations in a polynomial ring and require sampling from a discrete Gaussian distribution. To design a discrete Gaussian sampler that satisfies a negligible statistical distance to the accurate distribution, we analyze the Knuth-Yao random walk, and propose an algorithm that is fast and lightweight. For efficient polynomial multiplication, we apply the number theoretic transform. From these primitives we design a compact coprocessor that takes only $20/9\mu s$ to compute encryption/decryption on a Xilinx Virtex VI FPGA.

Homomorphic function evaluation is very slow in software due to its arithmetic involving very large polynomials with large coefficients. We design an FPGA-based accelerator for the homomorphic encryption scheme `YASHE`. We observe that though the computation intensive arithmetic can be accelerated, the overhead of external memory access becomes a bottleneck. Then we propose a more practical scheme that uses a special module to assist homomorphic function evaluation in less time. With this module we can evaluate encrypted search roughly 20 times faster than the implementation without this module.

Beknopte samenvatting

Het enorme netwerk van geïnterconnecteerde apparaten, gaande van kleine Radio frequentie identificatie tags (RFID) tot krachtige desktopcomputers, genereert een enorme hoeveelheid aan informatie. Asymmetrische cryptografie (PKC) speelt een cruciale rol bij de beveiliging van dit netwerk. In deze thesis focussen we op het efficiënt implementeren van PKC om aan de toekomstige beveiligingsuitdagingen het hoofd te bieden.

Het doel is het beveiligen van in middelen beperkte geconnecteerde apparaten. Hiertoe ontwikkelen we een compacte elliptische curve coprocessor voor een 283 bit Koblitz curve die een veiligheidsniveau van 140 bits biedt. We optimaliseren de scalaire transformatie, een belangrijk onderdeel van de punt vermenigvuldiging en introduceren een lichtgewicht tegenmaatregel tegen tijds en nevenkanaals aanvallen. De coprocessor verbruikt enkel 4.3 KGE.

In het tweede deel van de thesis onderzoeken we de verschillende aspecten van het implementeren van "post-quantum" PKC en homomorfische encryptie algoritmes wiens veiligheid gebaseerd is op de moeilijkheid van het oplossen van het ring-LWE probleem. Deze cryptografische algoritmes voeren rekenkundige operaties uit in een polynomische ring en vereisen het monsteren van discrete gaussische distributies. Om een discrete gaussische bemonsteraar te ontwerpen die een te verwaarloosbare statistische afstand heeft tot de perfecte distributie analyseren we de Knuth-Yao willekeurige wandeling. We stellen een algoritme voor dat zowel snel en compact is. Om een efficiënte polynomische vermenigvuldiging te bekomen passen we de getal theoretische transformatie toe. We vertrekken vanuit deze technieken voor het ontwerpen van een compacte coprocessor die slechts $20/9\mu s$ nodig heeft voor het uitvoeren van een en-/ decryptie op een Xilinx Virtex VI FPGA.

Het evalueren van homomorfische functies verloopt zeer traag in software door de rekenkundige technieken nodig voor het werken met polynomen met zeer grote coëfficiënten. We ontwerpen een FPGA gebaseerde versneller voor het

homomorfische encryptie algoritme `YASHE`. We bemerken dat de rekenkundige operaties versneld kunnen worden, maar dat het raadplegen van extern geheugen een bottleneck wordt. Vervolgens stellen we een praktischer algoritme voor dat gebruikt maakt van een speciale module dat ons in staat stelt de homomorfische encryptie sneller te evalueren. Met deze module kunnen we de geëncrypteerde opzoeking met een factor 20 versnellen in vergelijking met een geëncrypteerde opzoeking zonder de speciale module.

Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic and Logic Unit
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BLISS	Bimodal Lattice Signature Scheme
BRAM	Block RAM
CDT	Cumulative Distribution Table
CRT	Chinese remainder theorem
CVP	Closest Vector Problem
DDG	Discrete Distribution Generating
DES	Data Encryption Standard
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
DPA	Differential Power Analysis
DRAM	Distributed RAM
DRU	Division and Rounding Unit
DSP	Digital Signal Processor
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
FF	Flip Flop
FFT	Fast Fourier Transform
FHE	Fully Homomorphic Encryption

FPGA	Field Programmable Gate Array
FV	Fan-Vercauteren
IoT	Internet of Things
LPR	Lindner-Peikert-Regev
LUT	Lookup Table
LWE	Learning-With-Errors
MAC	Multiply and Accumulate
MPC	Multiparty Computation
NTT	Number Theoretic Transform
PALU	Polynomial Arithmetic and Logic Unit
PIR	Private Information Retrieval
PKC	Public Key Cryptography
RFID	Radio Frequency Identification
ring-LWE	Ring-Learning-With-Errors
SHE	Somewhat Homomorphic Encryption
SIMD	Single Instruction Multiple Data
SPA	Simple Power Analysis
SVP	Shortest Vector Problem
YASHE	Yet Another Somewhat Homomorphic Encryption

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the advent of the internet, our world has become more and more connected every day. The International Telecommunications Union reports [132] that the number of internet users has increased from 400 million in 2000 to 3.2 billion in 2015. This growth rate is expected to be faster in the future as a result of internet penetration in the developing nations. The Internet of Things (IoT) is a network of connected devices ranging from powerful personal computers and smart phones to low-cost passive RFID tags. These devices are capable of computing together and exchanging information with or without human intervention and are present in many areas of our life such as smart homes, smart grids, intelligent transportation, smart cities. By 2020 there will be 21 billion IoT devices [1]. These connected devices could upload their data or even outsource costly computational tasks to a cloud server. A cloud server is a very powerful device with huge storage and computation capability. Indeed cloud computing and IoT are tightly coupled.

In this connected world, our daily life applications such as email, social networks, e-commerce, online banking and several others generate and process massive amounts of information every day [151]. Snowden's revelation [134] in 2013 has brought security and privacy issues into the spotlight of media coverage. Google, Facebook and other leading internet companies are facing increasing pressures from government spying agencies to reveal information about the users. Now users are more concerned about security and privacy than before. Therefore it is of vital importance to protect digital information by incorporating confidentiality, integrity, and data availability.

Cryptography is the science of protecting digital information. In a broader sense, our present day cryptography schemes can be split into two branches: the

symmetric-key cryptography schemes and the public-key cryptography schemes. In a symmetric-key cryptography application, the two communicating parties use a common key to protect their information. The existing symmetric-key cryptography schemes are computationally very fast. However their security is based on the assumption that the two parties agree on a common key secretly before initiating the communication. Public-key cryptography is free from this assumption as there is no need for a common key. This feature makes public-key cryptography schemes very attractive despite their slower performance. In practice, most cryptographic protocols use both symmetric-key cryptography and public-key cryptography in tandem: a public-key cryptography scheme is used to agree on a common key and then a symmetric-key cryptography scheme is used to secure a large amount of digital information. In this research we concentrate only on public-key cryptography.

The most widely used public-key cryptography schemes are the RSA cryptosystem [117] and the elliptic-curve cryptosystem (ECC) [69]. Security of the RSA and elliptic-curve cryptosystems is based on the hardness of the integer factorization problem and elliptic-curve discrete logarithm problem (ECDLP) respectively. Although the RSA cryptosystem is conceptually a simple scheme, the main disadvantages are its large key size and slow private key operations. In comparison, ECC requires much smaller key size as the ECDLP problem is much harder to break than the integer factorization problem [12]. Nevertheless ECC requires expensive computation. To improve the efficiency of ECC, numerous proposals have been published in the literature in the past three decades. Such proposals [54] include choice of finite field, choice of elliptic-curve, optimizations in the finite field arithmetic, design of efficient algorithms, and finally tailoring the algorithms for applications and platforms. Designing ECC for lightweight IoT applications with low resources has been an extremely active research field in recent years [11, 13, 18, 55, 73, 74, 100, 144, 145]. These proposals focus predominantly on 163-bit elliptic-curves which provide medium security level of about 80 bits. However recent advances in the cryptanalysis have brought the 80-bit security level too close to call *insecure* for applications that require long term security. For e.g., recently FPGA-based hardware accelerators have been used to solve a 117.35-bit ECDLP on an elliptic-curve over $\mathbb{F}_{2^{127}}$ [17]. Moreover the National Institute of Standards and Technology (NIST) has recommended phasing out usage of 160-bit elliptic-curve cryptography by the end of the year 2010 [93]. In the first part of this thesis we address this problem by designing a lightweight ECC coprocessor using a high security 283-bit Koblitz curve.

Public-key cryptography is an ever evolving branch of cryptography. With our present day computers, RSA and ECC schemes are considered secure when the key size is sufficiently large and countermeasures against side channel and fault attacks are enabled. However this situation changes in the domain of quantum

Table 1.1: NIST recommended approximate key length for bit-security [12]

Algorithm	80-bit	112-bit	140-bit.
RSA	1024	2048	3072
ECC	160	224	256

computing. In 1994, Shor [126] designed a quantum algorithm that renders the above schemes insecure. Though there is no known powerful quantum computer today, several public and private organizations are trying to build quantum computers due to its potential applications. In 2014 a BBC News article [14] reports that the NSA is building a code cracking quantum computer. Post-quantum cryptography is a branch of cryptography that focuses on the design and analysis of schemes that are secure against quantum computing attacks. Beside the scientific community, several standardization bodies and commercial organizations are considering post-quantum cryptography. In 2016 NIST recommended a gradual shift towards post-quantum cryptography [95] and called for a standardization process for post-quantum public-key cryptography schemes. Recently Google has integrated a post-quantum cryptography scheme called Frodo [19] in 1% of all Chrome browsers for experimentation.

Amongst several candidates for post-quantum public-key cryptography, lattice-based cryptography appears to be the most promising because of its computational efficiency, strong security assurance, and support for a wide range of applications. Lattice-based cryptography has become a hot research topic in this decade since the introduction of the Learning-With-Errors (LWE) problem in 2009 [113] and its more efficient ring variant, the ring-LWE problem in 2010 [85]. Till 2012 almost all of the literature addressed the theoretical aspects of LWE and ring-LWE-based cryptography and very little was known about the implementation feasibilities and performance aspects. This motivated us to investigate implementation aspects of ring-LWE-based public-key cryptography in this research.

1.1 Summary of the thesis

In a broader sense the thesis investigates implementation aspects of next generation public-key cryptography on hardware platforms. The organization of the thesis is illustrated in Fig 1.1. In Chapter 3 the thesis takes account of the present day security challenges and studies efficient methods for implementing high security elliptic-curve cryptography on resource-constrained IoT platforms.

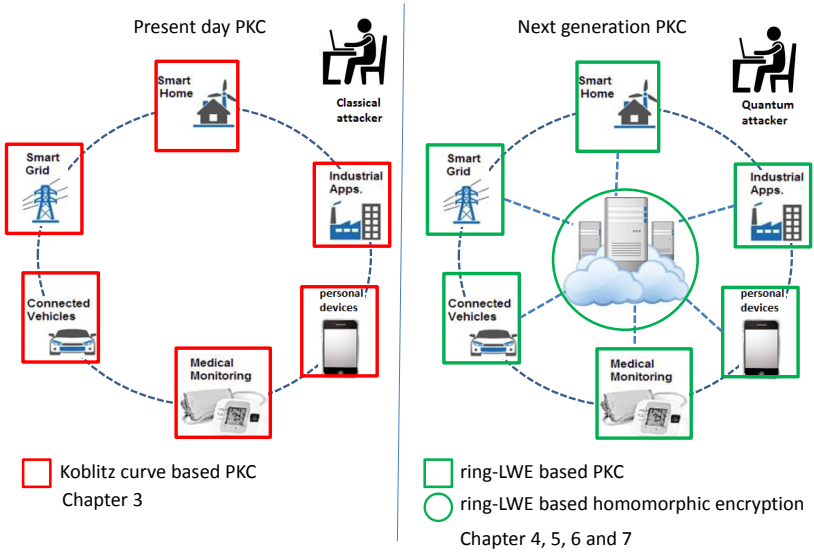

Figure 1.1: Structure of the thesis

The remaining part of the thesis investigates implementation feasibility of next generation ring-LWE-based post-quantum cryptography on terminal devices and cloud servers. We describe the contributions of each chapter below.

- **Chapter 2** We give a brief introduction to the mathematical background of the elliptic-curve discrete logarithm problem and the ring-LWE problem. Then we describe the public-key cryptosystems that we consider for implementation in this research.

- **Chapter 3** In this chapter we focus on an efficient implementation of a Koblitz curve point multiplier targeting a high security level. Koblitz curves are a class of computationally efficient elliptic-curves that offer fast point multiplications if the scalars are given as specific τ-adic expansions. This needs conversion from integer scalars to equivalent τ-adic expansions. We propose the first lightweight variant of the conversion algorithm and introduce the first lightweight implementation of Koblitz curves that includes the scalar conversion. We also include countermeasures against side-channel attacks making the coprocessor the first lightweight

coprocessor for Koblitz curves that includes a set of countermeasures against timing attacks, SPA, DPA and safe-error fault attacks.

This chapter's contents are derived from two publications.

Accelerating scalar conversion for Koblitz curve cryptoprocessors on hardware platforms [118] by Sujoy Sinha Roy, Junfeng Fan, and Ingrid Verbauwhede, published in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2015.

Lightweight Coprocessor for Koblitz Curves: 283-Bit ECC Including Scalar Conversion with only 4300 Gates [119] by Sujoy Sinha Roy, Kimmo Järvinen, and Ingrid Verbauwhede, published in Cryptographic Hardware and Embedded Systems (CHES 2015).

- **Chapter 4** The focus of this chapter is to design a high-precision and computationally efficient discrete Gaussian sampler for lattice-based post-quantum cryptography. Discrete Gaussian sampling is an integral part of many lattice-based cryptosystems such as public-key encryption schemes, digital signature schemes and homomorphic encryption schemes. We choose the Knuth-Yao sampling algorithm and propose a novel implementation of the algorithm based on an efficient traversal of the discrete distribution generating (DDG) tree. We investigate various optimization techniques to achieve minimum area and computation time. Next we study timing and power attacks on the Knuth-Yao sampler and propose a random shuffling countermeasure to protect the Gaussian distributed samples against such attacks.

  This chapter's contents are derived from two major publications.

  High precision discrete Gaussian sampling on FPGAs [123] by Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede, published in Selected Areas in Cryptography (SAC 2013).

  Compact and Side Channel Secure Discrete Gaussian Sampling [121] by Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede, published in IACR Cryptology ePrint Archive 2014.

- **Chapter 5** In this chapter we design an efficient and compact ring-LWE-based public-key encryption processor. The encryption processor is composed of two main components: a polynomial arithmetic unit and a discrete Gaussian sampler. For the discrete Gaussian sampling, we use the Knuth-Yao sampler from Chapter 4. For polynomial multiplication, we apply the Number Theoretic Transform (NTT). We propose three optimizations for the NTT to speed up computation and reduce resource requirement. Finally, at the system level, we also propose an optimization of the ring-LWE encryption that reduces the number of NTT operations.

We use these computational optimizations along with several architectural optimizations to design an instruction-set ring-LWE public-key encryption processor on FPGA platforms.

This chapter's content is derived from the publication

Compact Ring-LWE Cryptoprocessor [122] by Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede, published in Cryptographic Hardware and Embedded Systems (CHES 2014).

- **Chapter 6** This chapter of the thesis focuses on the implementation of hardware accelerators for ring-LWE-based homomorphic function evaluation. We design a modular hardware architecture for all building blocks required to instantiate the somewhat homomorphic encryption (SHE) scheme YASHE. We investigate efficient arithmetic to parallelize the costly polynomial arithmetic.

  The content of this chapter is derived from the publication

  Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation [120] by Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede, published in Cryptographic Hardware and Embedded Systems (CHES 2015).

- **Chapter 7** We observe that even with a very powerful hardware accelerator, we are unable to transform homomorphic encryption into a practical solution for private cloud computing. So, we interpolate between fully homomorphic encryption (FHE) and multiparty computation (MPC), and propose a more practical scheme to perform homomorphic evaluations of arbitrary depth with the assistance of a special module called *recryption box*. To demonstrate the practicality of the proposal, we design the recryption box on a Xilinx Virtex 6 FPGA board and evaluate the performance of an encrypted search operation.

  The content of this chapter is derived from the publication

  Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search by Sujoy Sinha Roy, Frederik Vercauteren, Jo Vliegen, and Ingrid Verbauwhede, accepted in IEEE Transactions on Computers.

- **Chapter 8** This chapter concludes the thesis and formulates future works.

Chapter 2

Background

2.1 Introduction to public-key cryptography

In this chapter we review the concept of public-key cryptography (PKC) and then describe two PKC schemes namely, elliptic-curve cryptography and lattice-based cryptography. PKC was introduced by Diffie and Hellman in 1976 [35]. In a PKC scheme, a user, say Bob, has a pair of keys: a widely disseminated *public-key* and a secret *private-key*. To send messages to Bob, Alice and other users use Bob's public-key. Only Bob can recover the messages from the ciphertexts by using his private-key. The basic concept of the public-key encryption is shown in Fig. 2.1.

The security of a PKC scheme is based on the assumption that it is computationally in-feasible to compute the private-key from the public-key. Such security assumption is assured by computationally hard mathematical problems such as the integer factorization problem, the discrete logarithm problem, the

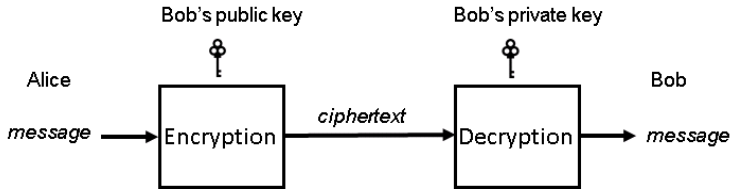

Figure 2.1: Basic concept of public-key encryption

elliptic-curve discrete logarithm problem, and several hard problems defined over lattices etc. In this research we implement a set of PKC schemes based on the elliptic-curve discrete logarithm problem and lattice problems.

Homomorphic encryption allows computations to be performed on encrypted data. Due to the homomorphism, equivalent computations are automatically performed on the plaintext data. Thus with homomorphic operations, users can upload their encrypted data to a powerful cloud service and still perform computations in the cloud on the encrypted data. A homomorphic encryption scheme is an augmented encryption scheme with two additional functions `HE.Add`() and `HE.Mult`() to add or multiply on ciphertexts, that result in a ciphertext encrypting the sum or respectively the product of the underlying plaintexts. Fig. 2.2 shows the application of homomorphic encryption in cloud computing.

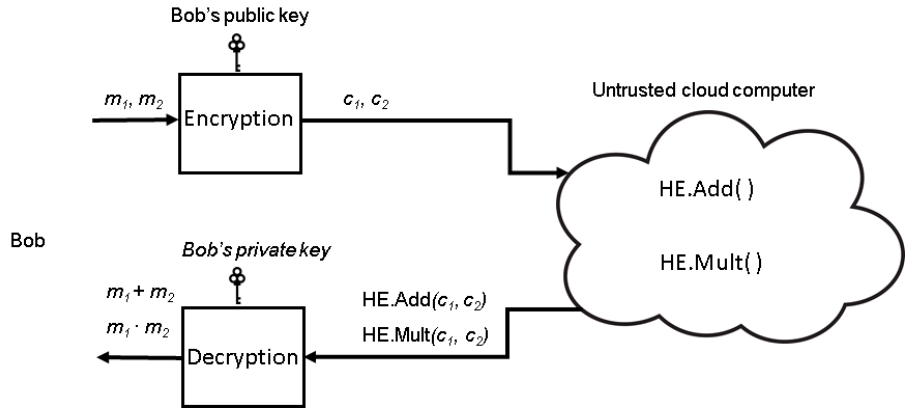

Figure 2.2: Homomorphic encryption in cloud computing

The chapter is organized as follows: in the remaining part of this section we define the elliptic-curve discrete logarithm problem and the well-known lattice problems. In Sect. 2.2 we introduce binary extension fields and Koblitz curves. The next section briefly describes the field primitives and a public-key encryption scheme that uses elliptic-curve cryptography. Cryptography schemes based on lattice problems are described in Sect. 2.4. The section also briefly describes the building blocks for implementing the lattice-based schemes. The final section gives a summary.

2.1.1 The elliptic-curve discrete logarithm problem

In 1986 Koblitz [69] and Miller [89] independently proposed cryptography using elliptic-curves, and since then elliptic-curve cryptography (ECC) has become very popular for designing fast public-key schemes on various platforms.

Definition 2.1.1. (Elliptic-curves [54]). An elliptic-curve E over a field $\mathbb{K}$ is defined by a so-called Weierstrass equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \ , \tag{2.1}$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ and $\Delta \neq 0$, where Δ is the discriminant, defined as follows:

$$\begin{aligned}
\Delta &= -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\
d_2 &= a_1^2 + 4a_2 \\
d_4 &= 2a_4 + a_1a_3 \\
d_6 &= a_3^2 + 4a_6 \\
d_8 &= a_1a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \ .
\end{aligned}$$

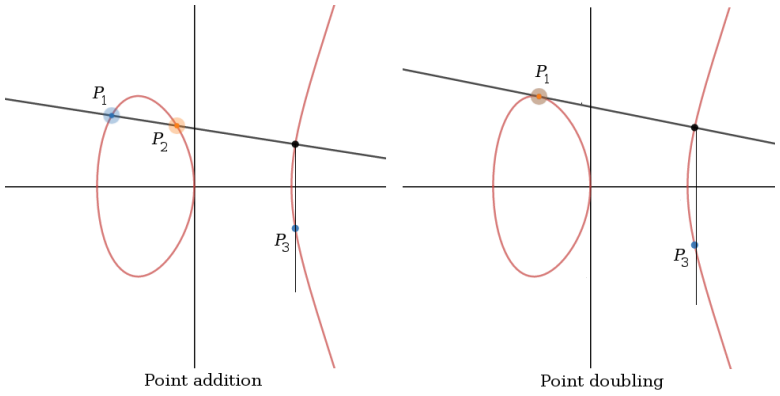

Figure 2.3: Geometric representation of point addition and doubling using the chord-and-tangent rule

Let $P(x, y)$ represents a point on E with coordinates (x, y). All points on the curve and a special point ∞ known as the point at infinity, form a group under the elliptic-curve addition rule in $\mathbb{K}$. Let us denote the group by $E(\mathbb{K})$. Let two

points on E be $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. Their sum is the point $P_3(x_3, y_3)$ on the curve. The addition rule uses the *chord-and-tangent* method for adding two points on the curve (Fig. 2.3). The method computes additions, subtractions, multiplications, and inversions in $\mathbb{K}$ (see Sect. 2.2).

By using the point addition and doubling operations, the well-known *double-and-add* method computes a scalar multiple k of $P(x, y)$ which is again a point $Q = k \cdot P$ on E. This computation is called *scalar multiplication* or *point multiplication*. Alg. 1 shows the double-and-add approach for point multiplication.

Algorithm 1: *Double-and-add* method for point multiplication [54]

Input: Scalar $k = \sum_{i=0}^{l-1} k_i 2^i$ and a point P on E
Output: Point $Q = k \cdot P$ on E

```
1 begin
2 |   Q ← ∞;
3 |   for i = l − 1 downto 0 do
4 |   |   Q ← 2Q ;
5 |   |   if k_i = 1 then
6 |   |   |   Q ← Q + P ;
7 |   |   end
8 |   end
9 end
```

Here P is called the base point and Q is called the scalar multiple of the base point.

Definition 2.1.2. (The elliptic-curve discrete logarithm problem (ECDLP)). For a given base point P on the curve E and a scalar multiple $Q = k \cdot P$, find the scalar k.

The ECDLP is considered to be a hard problem when the order of the base point has a large prime factor p. Advanced algorithms for solving ECDLP, such as Pollard's rho algorithm [102] has an expected time complexity $\mathcal{O}(\sqrt{p})$. A survey of the attack algorithms is provided by Galbraith and Gaudry [45]. The ECDLP has been used to construct key exchange schemes, public-key encryption schemes, and digital signature schemes [54].

2.1.2 Lattice problems

In this section we review hard problems defined over lattices. Such problems are a class of optimization problems and their conjectured intractability is the foundation of lattice-based public-key cryptography schemes.

Definition 2.1.3. (Lattice [58]). Let $\mathcal{V}$ be a set of n linearly independent vectors $v_0, \ldots, v_{n-1} \in \mathbb{R}^m$. The lattice $\mathcal{L}$ is the set of linear combinations of the vectors with coefficients in $\mathbb{Z}$.

$$\mathcal{L} = \{a_0 \cdot v_0 + \ldots + a_{n-1} \cdot v_{n-1}\} \text{ where } a_0, \ldots a_{n-1} \in \mathbb{Z} \ . \tag{2.2}$$

The set $\mathcal{V}$ is a *basis* of $\mathcal{L}$, n is its rank and m is its dimension. The lattice is called a full-rank lattice if $n = m$. Indeed a basis for $\mathcal{L}$ is any set of n independent vectors that generates $\mathcal{L}$, and there are infinitely many basis for $m \geq 2$. Any two such basis are related by an integer matrix with determinant equal to ± 1..

Definition 2.1.4. (Span [58]). The span of the basis of a lattice $\mathcal{L}$ with basis $\mathcal{V}$ is the collection of all linear combinations $\alpha_0 \cdot v_0 + \ldots + \alpha_{n-1} \cdot v_{n-1}$, where $\alpha_0, \ldots, \alpha_{n-1} \in \mathbb{R}$.

One parameter for $\mathcal{L}$ is the length of a shortest nonzero vector. The length of a vector v in $\mathcal{L}$ is defined by its Euclidean norm $\| v \|$. The length of shortest vector is denoted by λ_1, which is the smallest radius r such that the lattice points inside a ball of radius r span a space of dimension 1.

Definition 2.1.5. (Successive minima [112]). Let Λ be a lattice of rank n. For $i \in \{1, \ldots, n\}$, we define the ith successive minimum as

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

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

Definition 2.1.6. (The shortest vector problem (`SVP` [112])). Find a nonzero vector v in a lattice $\mathcal{L}$ such that $\| v \| = \lambda_1(\mathcal{L})$.

Ajtai [3] showed that the `SVP` with Euclidean norm is NP-hard for randomized reductions. The $\texttt{SVP}_\alpha$ is an α-approximation version of the SVP where one has to find a vector v_α in $\mathcal{L}$ such that $\| v_\alpha \| \leq \alpha \lambda_1(\mathcal{L})$. There is an absolute constant $\epsilon > 0$ so that the $\texttt{SVP}_\alpha$ is also NP-hard with $\alpha < 1 + 2^{-n^\epsilon}$ for randomized reductions [3].

Definition 2.1.7. (The closest vector problem (`CVP`) [58]). Given a vector $w \in \mathbb{R}^m$ that is not in $\mathcal{L}$, find a vector $v \in \mathcal{L}$ that is closest to w, i.e., find a vector $v \in \mathcal{L}$ that minimizes the Euclidean norm $\| w - v \|$.

Similar to the $\texttt{SVP}_\alpha$, the $\texttt{CVP}_\alpha$ is an α-approximation version of the `CVP` where one has to find a vector v_α such that $\| w - v_\alpha \| \leq \alpha \| w - v \|$. The $\texttt{CVP}_\alpha$ is a

generalization of the $\texttt{SVP}_\alpha$ and given an oracle for the $\texttt{CVP}_\alpha$ one can solve the $\texttt{SVP}_\alpha$ by making queries. The CVP is known to be NP-hard [88].

The SVP or CVP or their approximation versions are easy to solve if a basis comprised of orthogonal or nearly orthogonal and short vectors is known. Lattice reduction algorithms are a class of algorithms that aim to output such *good* basis from any given basis for a lattice. The LLL algorithm outputs an LLL-reduced basis in polynomial time but the with approximation factor C^n, where C is a small constant. Thus the LLL algorithm is very effective when the dimension n of the lattice is small. Algorithms that achieve close approximation (e.g. the AKS algorithm [4], the BKZ algorithm [125] etc.) run in exponential time. The inability of the lattice reduction algorithms to find a good basis in polynomial time is used as the foundation for the lattice-based cryptography schemes.

Construction of cryptographic schemes based on the hardness of lattice problems started with Ajtai's [2] seminal work where he showed average case to worst case reduction. This is of particular interest because the well-known number-theoretic problems such as the integer factorization or the ECDLP do not possess this feature. In 2005 Regev [111] introduced a new problem known as the learning with errors problem (LWE). Since its introduction, the LWE problem has become very popular for construction of a variety of schemes such as public-key encryption, key exchange, digital signature schemes and even homomorphic encryption schemes. The LWE problem is parametrized by the rank n of the lattice, an integer modulus q and an error distribution $\mathcal{X}$ over $\mathbb{Z}$. A secret vector $\mathbf{s}$ of rank n is chosen uniformly in $\mathbb{Z}_q^n$. Then samples are produced by selecting uniform random vectors $\mathbf{a}_i$ and error terms e_i from the error distribution $\mathcal{X}$ and by computing $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \in \mathbb{Z}_q$. The LWE distribution $A_{s,\mathcal{X}}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is defined as the set of tuples $(\mathbf{a}_i, b_i)$. The decision and search versions of the LWE problem are defined below.

Definition 2.1.8. (The decision LWE problem [111]). Distinguish with non-negligible advantage between a polynomial number of samples drawn from the LWE distribution $A_{s,\mathcal{X}}$ and the same number of samples drawn uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

Definition 2.1.9. (The search LWE problem [111]). Find the secret $\mathbf{s}$ given a polynomial number of samples from the LWE distribution $A_{s,\mathcal{X}}$.

Its hardness can be reduced to the hardness of the above mentioned lattice problems. In practice cryptosystems based on the LWE problem are slow as they require computations on large matrices with coefficients from $\mathbb{Z}_q$. There is a more practical variant of the LWE problem that is defined over polynomial rings and is called the *ring-LWE problem*.

The ring-LWE problem is a ring-based version of the LWE problem and was introduced by Lyubashevsky, Peikert and Regev in [85]. To achieve computational efficiency and to reduce the key size, ring-LWE uses special structured ideal lattices. Such lattices correspond to ideals in rings $R = \mathbb{Z}[\mathbf{x}]/\langle f \rangle$, where f is an irreducible polynomial of degree n. Let s be a secret uniformly random polynomial in $R_q = R/qR$. The ring-LWE distribution on $R_q \times R_q$ consists of polynomial tuples $(a_i(x), b_i(x))$, where the coefficients of a_i are chosen uniformly from $\mathbb{Z}_q$ and $b_i(x)$ is computed as a polynomial $a_i(x) \cdot s(x) + e_i(x) \in R_q$. Here e_i are error polynomials with coefficients sampled from an n-dimensional error distribution $\mathcal{X}$. The error distribution is typically a discrete Gaussian distribution. In some cases, e.g., for 2^k-power cyclotomics, this error distribution can be taken as the product of n independent discrete Gaussians, but in general $\mathcal{X}$ is more complex. One can construct s by sampling the coefficients from $\mathcal{X}$ instead of sampling uniformly without any security implications [85].

Definition 2.1.10. (The decision ring-LWE problem [85]). Distinguish with non-negligible advantage between a polynomial number of samples $(a_i(x), b_i(x))$ drawn from the ring-LWE distribution and the same number of samples generated by choosing the coefficients uniformly.

Definition 2.1.11. (The search ring-LWE problem [85]). Find the secret polynomial $s(x)$ given a polynomial number of samples drawn from the ring-LWE distribution.

In cases where f is a cyclotomic polynomial, the difficulty [85] of the search ring-LWE problem is roughly equivalent to finding a short vector in an ideal lattice composed of polynomials from R. Note that in the case of the LWE problem, the hardness was related to solving the NP-hard $\texttt{SVP}_\alpha$ over *general* lattices. Though no proof exists to show equivalence between the $\texttt{SVP}_\alpha$ for general lattices and ideal lattices, the two cases are presumed to be equally difficult. The computational efficiency using the ring-LWE problem is obtained at the cost of the above security assumption. Cryptographic schemes based on the ring-LWE problem are fast thanks to simple polynomial arithmetic [52].

2.2 Elliptic-curve cryptography over $\mathbb{F}_{2^m}$

In cryptography, elliptic-curves defined over finite fields are used. The most commonly used finite fields for ECC are prime fields and binary extension fields $\mathbb{F}_{2^m}$. For hardware implementations, elliptic-curves over $\mathbb{F}_{2^m}$ are preferred since they can be implemented easily on hardware and because they achieve faster speed than prime fields. In this research we restrict ourselves to elliptic-curves over $\mathbb{F}_{2^m}$.

Definition 2.2.1. (Elliptic-curves over binary fields). The curve equation is

$$E : y^2 + xy = x^3 + ax^2 + b \ , \tag{2.3}$$

where the curve constants a and b are in $\mathbb{F}_{2^m}$.

For elliptic-curves defined over binary fields, the point addition and doubling rules are defined below.

Point addition For $P_1(x_1, y_1) \neq P_2(x_2, y_2)$, the equation for $P_3(x_3, y_3)$ is as follows.

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{and} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1 \ . \tag{2.4}$$

Point doubling For $P_1(x_1, y_1) = P_2(x_2, y_2)$, the equation for $P_3(x_3, y_3)$ is as follows.

$$x_3 = \lambda^2 + \lambda + a \quad \text{and} \quad y_3 = x_1^2 + \lambda x_3 + x_3 \ . \tag{2.5}$$

2.2.1 Koblitz curves

Koblitz curves introduced by Koblitz in [70] are a special class of elliptic-curves defined by the following equation:

$$y^2 + xy = x^3 + ax^2 + 1 \ , \quad a \in \{0, 1\} \ , \tag{2.6}$$

with points with coordinates $x, y \in \mathbb{F}_{2^m}$. Koblitz curves offer efficient point multiplications because they allow trading computationally expensive point doublings to cheap Frobenius endomorphisms. Many standards use Koblitz curves including NIST FIPS 186-4 [94] which describes the (Elliptic-Curve) Digital Signature Standard (ECDSA) and defines five Koblitz curves NIST K-163, K-233, K-283, K-409, and K-571 over the finite fields $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, and $\mathbb{F}_{2^{571}}$, respectively.

The Frobenius endomorphism for a point $P \in E$ is given by $\phi(P) = (x^2, y^2)$. For Koblitz curves, it holds that $\phi(P) \in E$ for all $P \in E$. Koblitz showed that $\phi^2(P) - \mu\phi(P) + 2P = \infty$ for all $P \in E$, where $\mu = (-1)^{1-a}$ [70]. Consequently, the Frobenius endomorphism can be seen as a multiplication by the complex number $\tau = (\mu + \sqrt{-7})/2$ [70].

Let the ring of polynomials in τ with integer coefficients be denoted by $\mathbb{Z}[\tau]$. For any element $u = u_{l-1}\tau^{l-1} + \ldots + u_0 \in \mathbb{Z}[\tau]$ with $u_i \in \{0, 1\}$, we can multiply any base point P by u as follows.

$$[u_{l-1}\tau^{l-1} + \ldots + u_0]P = [u_{l-1}]\tau^{l-1}P + \ldots + [u_0]P \ .$$

The point multiplication performs only point additions and Frobenius operations. Since the Frobenius operation is cheap, point multiplication by $u \in \mathbb{Z}[\tau]$ is faster than a point multiplication by an integer scalar of the same length. However, the ECDLP is defined for integer scalars only. Solinas showed that using the relation $-\tau^2 + \mu\tau = 2$, it is possible to map an integer scalar into an element in $\mathbb{Z}[\tau]$ with binary coefficients [129]. Such a representation is called a τ-adic expansion. Representing an integer scalar k as a τ-adic expansion $t = \sum_{i=0}^{\ell-1} t_i \tau^i$ allows computing point multiplications with a Frobenius-and-add algorithm, which is similar to the double-and-add algorithm except that point doublings are replaced by Frobenius endomorphisms.

Length reduction during integer to τ-adic conversion. If a τ-adic representation is computed directly from the integer scalar k, then the length l of the τ-adic representation is approximately two times the bit-length of k. This expansion in length is a problem since it doubles the number of point additions. Solinas showed [129] that if the integer scalar is expressed as $k = \lambda(\tau^m - 1) + \rho$, then the point multiplication $k \cdot P(x, y)$ turns into $\rho \cdot P(x, y)$ as for Koblitz curves $(\tau^m - 1)P(x, y) = \infty$. The good thing is that now a τ-adic representation of ρ has length roughly equal to m [129]. The computation of ρ from k is called the scalar reduction operation.

There are several methods for performing scalar reductions. The *lazy reduction* method proposed by Brumley and Järvinen [23] uses divisions by τ.

Theorem 1. (Division by τ). Any element $\alpha = (d_0 + d_1\tau) \in \mathbb{Z}[\tau]$ is divisible by τ if and only if d_0 is even. The result of the division when stored in (d_0, d_1), becomes

$$(d_0, d_1) \leftarrow (\mu d_0/2 + d_1, -d_0/2) \ .$$

Since the division by τ can be performed by simple shift and addition operations, the scalar reduction method of Brumley and Järvinen is suitable for lightweight hardware implementations. In Chap. 3 we use this scalar reduction method for our hardware implementation. The steps to perform point multiplication on Koblitz curves are shown in Fig. 2.4.

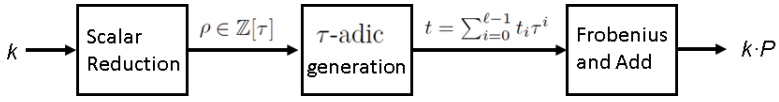

Figure 2.4: Computation flow in point multiplication on Koblitz curves

ECIES. As an example of ECC-based schemes, we describe the Elliptic-Curve Integrated Encryption Scheme (ECIES). The scheme was invented by Abdalla, Bellare and Rogaway [1] for the purpose of public-key encryption. Besides an elliptic-curve point multiplier, the ECIES scheme uses a key derivation function (KDF), a symmetric-key encryption/decryption algorithm ENC/DEC, and a message authentication code (MAC). The ECIES scheme is described as follows.

1. ECIES.KeyGen(E): For a public base point P on E, choose a random private-key k and compute the public-key $Q = k \cdot P$.
2. ECIES.Encrypt(Q, m): Randomly generate an integer r and compute the point multiplications $R_1 = r \cdot P$ and $R_2 = r \cdot Q$. Generate $(k_1, k_2) \leftarrow \mathtt{KDF}(x_{R_2}, R_1)$ where x_{R_2} is the x-coordinate of R_2. Compute the ciphertext $c \leftarrow \mathtt{ENC}(k_1, m)$ and compute the tag $t \leftarrow \mathtt{MAC}(k_2, c)$. Output R_1, c and t.
3. ECIES.Decrypt(k, R_1, c, t): Compute $R_3 = k \cdot R_1$ and then use the x-coordinate of R_3 i.e., x_{R_3} as the shared secret. Note that $R_3 = k \cdot R_1 = kr \cdot P = r \cdot Q = R_2$. Hence $x_{R_3} = x_{R_2}$. Now follow the same steps as described in ECIES.Encrypt and compute (k_1, k_2) as the keys for the DEC and the MAC. Now it is trivial to recover m from c using $\mathtt{DEC}(k_1, c)$ and then validate the recovered message.

From the above scheme we see that the elliptic-curve point multiplication plays a central role in the ECIES scheme. In this research, we restrict ourselves to the implementation of the elliptic-curve point multiplication primitive.

2.3 Primitives for arithmetic in $\mathbb{F}_{2^m}$

Let $f(x) = x^m + \bar{f}(x)$ be the irreducible binary polynomial of degree m for the binary extension field $\mathbb{F}_{2^m}$. There are two popular ways to represent the field elements: the polynomial basis and the normal basis representations. In this research we use the polynomial basis representation. In this representation an element $a \in \mathbb{F}_{2^m}$ is represented as a polynomial $a(\theta) = \sum_{i=0}^{m-1} a_i \theta^i$ where the coefficients a_i are from $\mathbb{F}_2$ and θ is a root of $f(x)$. Addition or subtraction of two elements is coefficient-wise addition or subtraction in $\mathbb{F}_2$. In the following we review the field reduction, multiplication, squaring and inversion operations.

2.3.1 Reduction

In the polynomial basis representation when two field elements are multiplied, the result, say $c(\theta)$, is a polynomial of degree at most $2m-2$. Since $f(x) = x^m + \bar{f}(x)$,

we have $\theta^m = \bar{f}(\theta)$ in $\mathbb{F}_{2^m}$. Note that $\bar{f}(\theta)$ has a degree at most $m-1$, and hence θ^m gets reduced to $\bar{f}(\theta)$. A naive approach to reduce $c(\theta)$ is to reduce the coefficients of θ^i sequentially for all $i \in [m, 2m-2]$. This method is slow due to its sequential nature. A speedup is possible by reducing more than one coefficients in every iteration.

We can do a lot better if $f(x)$ is sparse. NIST has recommended $\mathbb{F}_{2^m}$ fields with irreducible polynomials having three or five nonzero coefficients [92]. For such sparse irreducible polynomials, each coefficient of the reduced result can be expressed as a boolean expression of a few input coefficients. Hence the output coefficients can be computed directly by evaluating the small boolean expressions. In this research we use the NIST recommended K-283 curve over $\mathbb{F}_{2^{283}}$ with $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$. The steps using this irreducible polynomial are shown in Alg. 2.43 in [54].

2.3.2 Multiplication

The field multiplication in the polynomial basis representation is the multiplication of two polynomials, followed by a reduction by $f(x)$. For two elements a and $b \in \mathbb{F}_{2^m}$ the result is $\sum_{i=0}^{m-1} b(\theta) a_i \theta^i \mod f(x)$. There are several ways to compute the polynomial multiplication efficiently. A detailed description of these methods can be found in [54].

The naive method is the classical shift-and-add method for polynomial multiplication. This method has a quadratic time complexity. The time requirement can be reduced by a factor by processing the operands in a word-serial way. An efficient way to perform word-serial processing is known as the *comb method* [54].

For fast computation, the Karatsuba method is the most efficient one thanks to its $\mathcal{O}(m^{\lg_2 3})$ time complexity. The Karatsuba method splits each input operand into two polynomials of length $\lceil m/2 \rceil$. For e.g., a is split into two half-size polynomials a_h and a_l such that $a = a_h \theta^{\lceil m/2 \rceil} + a_l$. After this splitting, the multiplication is performed as

$$a \cdot b = a_h \cdot b_h \theta^{2\lceil m/2 \rceil} + \big[(a_h + a_l) \cdot (b_h + b_l) + a_h \cdot b_h + a_l \cdot b_l\big]\theta^{\lceil m/2 \rceil} + a_l \cdot b_l \ .$$

In a similar fashion, each of the small multiplications can also be split into three smaller multiplications. This gives a recursive algorithm for computing multiplication. The Karatsuba method achieves good performance for bit-parallel implementation on hardware platforms [109]. However the recursive structure of the algorithm and additional storage requirement for the partial products are costly for lightweight implementations. In this research we use the word-serial comb method [54] to perform field multiplication.

2.3.3 Squaring

A squaring in $\mathbb{F}_{2^m}$ can be computed much faster than a multiplication. In the polynomial basis representation the square of $a = \sum_{i=0}^{m-1} a_i\theta^i$ is $a^2 = \sum_{i=0}^{m-1} a_i\theta^{2i}$. The squaring operation spreads out the input coefficients by inserting zeros between each input coefficients. On hardware platforms, the spreading out of the input bits can be implemented free of cost. The only cost is due to the modular reduction by $f(x)$. Still this cost is small as we use a sparse $f(x)$.

2.3.4 Inversion

The inverse of an element $a \in \mathbb{F}_{2^m}$ is the unique element, denoted as $a^{-1} \in \mathbb{F}_{2^m}$, such that $a \cdot a^{-1} \equiv 1 \mod f(x)$. Inversion is considered to be the costliest field operation. The most commonly used methods are based on the Extended Euclidean Algorithm (`EEA`) or the Fermat's Little Theorem (`FLT`) [54].

The `EEA` computes the greatest common divisor (`GCD`) of two polynomials a and b by finding two polynomials g and h such that $a \cdot g + b \cdot h = d$ where $d = \texttt{GCD}(a, b)$. This property of the `EEA` is used to compute the inverse of an element. Since $f(x)$ is irreducible in $\mathbb{F}_{2^m}$, the `GCD` of a and f is always one. Hence the `EEA` computes g and h such that $a \cdot g + f \cdot h = 1$, and with this we get $a \cdot g \equiv 1 \mod f(x)$. Naturally $a^{-1} = g$ in $\mathbb{F}_{2^m}$.

Inversion using the `FLT` computes $a^{-1} = a^{2^m-2}$. The computation requires exponentiation of the input by $2^m - 2$. Itoh and Tsujii [61] used addition chains to compute the exponentiation efficiently. The advantage of the `FLT`-based method over the `EEA` is that it requires only multiplications and squarings for the exponentiation. For hardware implementations, the `FLT`-based inversion method is well suited as it can reuse the multiply and square primitives. We use the `FLT`-based inversion for our lightweight implementation.

2.4 Ring-LWE-based cryptography

The ring-LWE problem has been used to construct a wide range of schemes such as public-key encryption, key exchange, digital signature and homomorphic encryption schemes. In this research we deal with ring-LWE-based public-key encryption and homomorphic schemes. We review these schemes in the remaining part of this section.

2.4.1 The LPR public-key encryption scheme

An elegant public-key encryption scheme was constructed by Lyubashevsky, Peikert, and Regev in the full version of [85] based on the ring-LWE problem. The LPR encryption scheme performs simple polynomial arithmetic such as polynomial multiplications, additions and subtractions, along with sampling from an error distribution typically a discrete Gaussian distribution χ_σ with a small standard deviation σ. It uses a global polynomial $a \in R_q$. The key generation, encryption and decryption are as follows.

1. `LPR.KeyGen`(a): Choose two polynomials $r_1, r_2 \in R_q$ from $\mathcal{X}_\sigma$ and compute $p = r_1 - a \cdot r_2 \in R_q$. The public-key is (a, p) and the private-key is r_2. The polynomial r_1 is simply noise and is no longer required after key generation.

2. `LPR.Encrypt`(a, p, m): The message m is first encoded to $\bar{m} \in R_q$. In the simplest type of encoding scheme a message bit is encoded as $(q-1)/2$ if the message bit is 1 and 0 otherwise. Three noise polynomials $e_1, e_2, e_3 \in R_q$ are sampled from a discrete Gaussian distribution with standard deviation σ. The ciphertext then consists of two polynomials $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + \bar{m} \in R_q$.

3. `LPR.Decrypt`(c_1, c_2, r_2): Compute $m' = c_1 \cdot r_2 + c_2 \in R_q$ and recover the original message m from m' using a decoder. In the simplest decoding scheme the coefficients m'_i of m' are decodes as 1 if they are in the interval $(q/4, 3q/4)$, and as 0 otherwise.

A block level view of the LPR encryption and decryption is shown in Fig. 2.5.

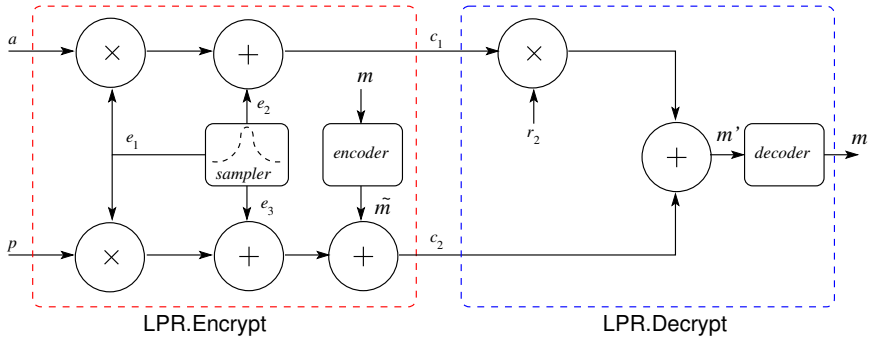

Figure 2.5: Block level `LPR.Encrypt` and `LPR.Decrypt`

2.4.2 Ring-LWE-based homomorphic encryption schemes

The beauty of the ring-LWE problem is that it is not restricted to encryption and signature schemes. It has been used to design efficient homomorphic encryption schemes. A ring-LWE-based homomorphic encryption scheme uses a basic ring-LWE encryption scheme and two additional functions `Add` and `Mult` to perform arithmetic operation on encrypted data. However in comparison to a simple ring-LWE encryption scheme, a homomorphic encryption scheme requires a much larger parameter set to support a desired multiplicative depth. In the next part we review two homomorphic encryption schemes namely FV and YASHE. Both schemes work in the polynomial ring $R = \mathbb{Z}[x]/\langle f(x)\rangle$ with $f(x) = \Phi_d(x)$, the d-th cyclotomic polynomial of degree $n = \varphi(d)$. A plaintext is an element in the ring $R_t = R/tR$ for some small modulus t. For binary operations t is taken as 2.

In the following part of this section we define two functions that are used to describe the FV and YASHE schemes. The key generation and the encryption operations in both FV and YASHE require sampling from two probability distributions defined on R, namely χ_{key} and χ_{err} respectively. The security is determined by the degree n of f, the size of q, and by the probability distributions. Following [85] one may sample the key and the error polynomials from a common distribution χ. Typically χ is a discrete Gaussian distribution χ_σ with a small standard deviation σ. However in practice some authors take the key as a polynomial with coefficients from a narrow set like $\{-1, 0, 1\}$.

Definition 2.4.1. ($\texttt{WordDecomp}_{w,q}(a)$). This function is used to decompose a ring element $a \in R_q$ in base w by splicing each coefficient of a. For $l = \lceil \log_w(q) \rceil$, this function returns $a_i \in R$ with coefficients in $(-w/2, w/2]$, where $a = \sum_{i=0}^{l-1} a_i w^i$.

Definition 2.4.2. ($\texttt{PowersOf}_{w,q}(a)$). This function scales an element $a \in R_q$ by the different powers of w. It is defined as $\texttt{PowersOf}_{w,q}(a) = (aw^i)_{i=0}^{l-1}$. The two functions can be used to perform a polynomial multiplication in R_q as

$$\langle \texttt{WordDecomp}_{w,q}(a), \texttt{PowersOf}_{w,q}(b) \rangle = a \cdot b \bmod q\,.$$

This expression has the advantage of reducing the noise during homomorphic multiplications, as the first vector contains small elements (in base w).

The FV homomorphic encryption scheme

The FV scheme was introduced by Fan and Vercauteren [43] in 2012. We briefly describe the functions used in the FV scheme. For details of the functions, interested readers are referred to the original paper [43].

1. $\texttt{FV.ParamsGen}(\lambda)$: For a given security parameter λ, choose a polynomial $\Phi_d(x)$, ciphertext modulus q and plaintext modulus t, and distributions χ_{err} and χ_{key}. Also choose the base w for $\texttt{WordDecomp}_{w,q}(\cdot)$. Return the system parameters $(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$. Following [43] we use a uniform signed binary distribution for χ_{key}. Additionally we set the plaintext modulus $t = 2$.

2. $\texttt{FV.KeyGen}(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$: Sample polynomial s from χ_{key}, sample $a \leftarrow R_q$ uniformly at random, and sample $e \leftarrow \chi_{err}$. Compute $b = [-(as + e)]_q$. The public-key consists of two polynomials $pk = \{b, a\}$ and the secret key is $sk = s$. The scheme uses another key called *relinearisation key* or **rlk** in the function `ReLin`. This key is computed as follows: first sample $\mathbf{a} \leftarrow R_q^l$ uniformly, then sample $\mathbf{e} \leftarrow \chi_{err}^l$, and then compute $\mathbf{rlk} = \{\mathbf{rlk}_0, \mathbf{rlk}_1\} = \{[\texttt{PowersOf}_{w,q}(s^2) - (\mathbf{e} + \mathbf{a} \cdot s)]_q, \mathbf{a}\} \in \{R_q^l, R_q^l\}$.

3. $\texttt{FV.Encrypt}(pk, m)$: First encode the input message $m \in R_t$ into a polynomial $\Delta m \in R_q$ with $\Delta = \lfloor q/t \rfloor$. Next sample the error polynomials $e_1, e_2 \leftarrow \chi_{err}$, sample u uniformly from the signed binary distribution, and, compute the two polynomials $c_0 = [\Delta m + bu + e_1]_q \in R_q$ and $c_1 = [au + e_2]_q \in R_q$. The ciphertext is the pair of polynomials $\mathbf{c} = \{c_0, c_1\}$.

4. $\texttt{FV.Decrypt}(sk, c)$: First compute a polynomial $\tilde{m} = [c_0 + sc_1]_q$. When $t = 2$, recover the plaintext message m by a decoding the coefficients of $\tilde{m}$. This decoding operation checks if the coefficient is in $(q/4, 3q/4)$ for a 1 bit and a 0 bit otherwise.

5. $\mathsf{FV.Add}(\mathbf{c_1}, \mathbf{c_2})$: For two ciphertexts $\mathbf{c_0} = \{c_{0,0}, c_{0,1}\}$ and $\mathbf{c_1} = \{c_{1,0}, c_{1,1}\}$, return $\mathbf{c} = \{c_{0,0} + c_{1,0}, c_{1,0} + c_{1,1}\}$.

6. $\mathsf{FV.Mult}(\mathbf{c}_1, \mathbf{c}_2, \mathbf{rlk})$: Compute $\tilde{\mathbf{c}}_{mult} = \{c_0, c_1, c_2\}$ where $c_0 = [\lfloor \frac{t}{q} \cdot c_{1,0} \cdot c_{2,0} \rceil]_q$, $c_1 = [\lfloor \frac{t}{q} \cdot (c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}) \rceil]_q$, and $c_2 = [\lfloor \frac{t}{q} \cdot c_{1,1} \cdot c_{2,1} \rceil]_q$. Next call the function $\texttt{ReLin}(\tilde{\mathbf{c}}_{\mathbf{mult}}, \mathbf{rlk})$.

7. $\texttt{FV.ReLin}(\tilde{\mathbf{c}}_{\mathbf{mult}}, \mathbf{rlk})$: Compute a relinearised ciphertext $\mathbf{c}' = \{[c_0 + \langle \texttt{WordDecomp}_{w,q}(c_2), \mathbf{rlk}_0 \rangle]_q, [c_1 + \langle \texttt{WordDecomp}_{w,q}(c_2), \mathbf{rlk}_1 \rangle]_q\}$.

The YASHE homomorphic encryption scheme

The YASHE scheme was introduced by Bos, Lauter, Loftus, and Naehrig [21] in 2013. The YASHE scheme is then defined as follows (full details can be found in the original paper [21]).

- $\texttt{YASHE.ParamsGen}(\lambda)$: For security parameter λ, choose a polynomial $\Phi_d(x)$, moduli q and t and distributions χ_{err} and χ_{key} attaining

security level λ. Also choose base w and return the system parameters $(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$.

- $\texttt{YASHE.KeyGen}(\Phi_d(x), q, t, \chi_{err}, \chi_{key}, w)$: Sample $f', g \leftarrow \chi_{key}$ and set $f = (tf' + 1) \in R_q$. If f is not invertible in R_q choose a new f'. Define $h = tgf^{-1} \in R_q$. Sample two vectors $\mathbf{e}$, $\mathbf{s}$ of $u + 1$ elements from χ_{err} and compute $\gamma = \texttt{PowersOf}_{w,q}(f) + \mathbf{e} + h\mathbf{s} \in R_q^{u+1}$ and output $(pk, sk, evk) = (h, f, \gamma)$. The key generation is based on the modified version of NTRU by Stehlé and Steinfeld [130].
- $\texttt{YASHE.Encrypt}(h, m)$: To encrypt a message $m \in R_t$ sample $s, e \leftarrow \chi_{err}$ and output the ciphertext $c = \Delta \cdot m + e + sh \in R_q$ with $\Delta = \lfloor q/t \rfloor$.
- $\texttt{YASHE.Decrypt}(f, c)$: Recover m as $m = \lfloor \frac{t}{q} \cdot [f \cdot c]_q \rceil \in R_t$ with $[\cdot]_q$ reduction in the interval $(-q/2, q/2]$.
- $\mathsf{YASHE.Add}(c_1, c_2)$: Return $c_1 + c_2 \in R_q$.
- $\mathsf{YASHE.KeySwitch}(c, evk)$: Return $\langle \texttt{WordDecomp}_{w,q}(c), evk \rangle \in R_q$
- $\mathsf{YASHE.Mult}(c_1, c_2, evk)$: Return $c = \mathsf{YASHE.KeySwitch}(c', evk)$ with $c' = [\lfloor \frac{t}{q} c_1 c_2 \rceil]_q \in R_q$.

2.5 Primitives for ring-LWE-based cryptography

From the descriptions of the ring-LWE-based public-key encryption and homomorphic encryption schemes, we see that the main primitives that we need are: discrete Gaussian sampling for the generation of the error polynomials, polynomial arithmetic unit for polynomial addition, subtraction and multiplication, and a division-and-round unit for computing homomorphic multiplications. These primitives are described as follows.

2.5.1 Discrete Gaussian sampler

Definition 2.5.1. (Discrete Gaussian distribution). The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}$ with mean 0 and standard deviation $\sigma > 0$ is defined by $D_{\mathbb{Z},\sigma}(E = z) = \frac{1}{S} e^{-(z)^2/2\sigma^2}$, where E is a random variable on $\mathbb{Z}$ and S is the normalization factor equal to $1 + 2\sum_{z=1}^{\infty} e^{-z^2/2\sigma^2}$ which is approximately $\sigma\sqrt{2\pi}$.

In the same way we can define a discrete Gaussian distribution $D_{\mathcal{L},\sigma}$ over a lattice $\mathcal{L}$. It assigns a probability proportional to $e^{-|\mathbf{v}|^2/2\sigma^2}$ to each element $\mathbf{v} \in \mathcal{L}$. Specifically when $\mathcal{L} = \mathbb{Z}^n$, the discrete Gaussian distribution is the product distribution of n independent copies of $D_{\mathbb{Z},\sigma}$.

Tail bound of a discrete Gaussian distribution: The tail of the Gaussian distribution is infinitely long and cannot be covered by any sampling algorithm. Indeed we need to sample up to a bound known as the *tail bound*. A finite tail-bound introduces a statistical difference with the true Gaussian distribution. The tail-bound depends on the maximum statistical distance allowed by the security parameters. As per Lemma 4.4 in [84], for any $c > 1$ the probability of sampling $\mathbf{v}$ from $D_{\mathbb{Z}^m,\sigma}$ satisfies the following inequality.

$$Pr(|\mathbf{v}| > c\sigma\sqrt{m}) < c^m e^{\frac{m}{2}(1-c^2)} \ . \tag{2.7}$$

Precision bound of a discrete Gaussian distribution: The probabilities in a discrete Gaussian distribution have infinitely long binary representations and hence no algorithm can sample according to a true discrete Gaussian distribution. Secure applications require sampling with high precision to maintain negligible statistical distance from the actual distribution. Let ρ_z denote the true probability of sampling $z \in \mathbb{Z}$ according to the distribution $D_{\mathbb{Z},\sigma}$. Assume that the sampler selects z with probability p_z where $|p_z - \rho_z| < \epsilon$ for some error-constant $\epsilon > 0$. Let $\tilde{D}_{\mathbb{Z},\sigma}$ denote the approximate discrete Gaussian distribution corresponding to the finite-precision probabilities p_z. The approximate distribution $\tilde{D}_{\mathbb{Z}^m,\sigma}$ corresponding to m independent samples from $\tilde{D}_{\mathbb{Z},\sigma}$ has the following statistical distance Δ to the true distribution $D_{\mathbb{Z}^m,\sigma}$ [39]:

$$\Delta(\tilde{D}_{\mathbb{Z}^m,\sigma}, D_{\mathbb{Z}^m,\sigma}) < 2^{-k} + 2mz_t\epsilon \ . \tag{2.8}$$

Here $Pr(|\mathbf{v}| > z_t : \mathbf{v} \leftarrow D_{\mathbb{Z}^m,\sigma}) < 2^{-k}$ represents the tail bound.

Methods for discrete Gaussian sampling

There are various methods for sampling from a discrete non-uniform (and also Gaussian) distribution [33]. Here we review most of these methods.

The rejection sampling is one of the simplest methods for sampling from a discrete nonuniform distribution. To sample from a target distribution T, the rejection sampling method first samples a value z from some easy proposal distribution. Then the sampled value is accepted with a probability proportional to $T_{\mathbb{Z},\sigma}(E = z)$. Though the method is simple in nature, in practice, rejection

sampling for a discrete Gaussian distribution is slow due to the high rejection rate for the sampled values that are far from the center of the distribution [52]. Moreover, for each trial, many random bits are required which is very time consuming on a constrained platform.

For continuous Gaussian distributions, the Ziggurat method is very efficient to minimize the rejection rate. Buchmann, Cabarcas, Göpfert, Hülsing and Weiden proposed a discrete version of the Ziggurat method [24] to sample from discrete Gaussian distributions. Similar to the well known continuous Ziggurat method, the discrete version divides the target distribution into several rectangles distributions. The rectangles are ordered with respect to the x-coordinates of their right edges. During a sampling operation, first a rectangle is randomly chosen, and then a random coordinate x-coordinate is generated within the rectangle. The random x-coordinate is accepted as the sample output if it is also within the rectangle that is just before the randomly chosen rectangle in the ordered list. When the condition is not satisfied, a random y coordinate is generated and a costly $exp()$ computation is performed. The authors showed that discrete Ziggurat could be a good choice to reduce memory requirement when the standard deviation is large.

The inversion sampling method first generates a random probability and then selects a sample value such that the cumulative distribution up to that sample point is just larger than the randomly generated probability. To implement discrete Gaussian sampling using the inversion method, the cumulative distribution function (CDF) table `CDT` is precomputed with necessary tail and precision bound. The table has the property $\texttt{CDT}[i+1] - \texttt{CDT}[i] = D_{\mathbb{Z},\sigma}(E = i)$. During a sampling operation, a random number r is generated uniformly and then the `CDT` table is searched to find an index z such that $\texttt{CDT}[z] \leq r < \texttt{CDT}[z+1]$. The output from the sampling operation is z. Note that the bit width of the random number r should be equal to the precision bound of the distribution. As a result, this method requires a large number of random bits.

The Knuth-Yao sampling [68] uses a random walk model to generate samples from a known nonuniform discrete distribution. For the known distribution a rooted binary tree known as the discrete distribution generating (DDG) tree is constructed. The DDG tree consists of two types of nodes: intermediate nodes (I) and terminal nodes. A terminal node contains a sample point, whereas an intermediate node generates two child nodes in the next level of the DDG tree. During a sampling operation a random walk is performed starting from the root of the DDG tree. For every jump from one level of the DDG tree to the next level, a random bit is used to determine a child node. The sampling operation terminates when the random walk hits a terminal node. The value of the terminal node is the value of the output sample point. The

Knuth-Yao algorithm performs sampling from non-uniform distributions using a near-optimal number of random bits.

A detailed comparative analysis of different sampling methods can be found in [39]. In this research we use the Knuth-Yao method to design a discrete Gaussian sampler for the above mentioned public-key encryption and homomorphic encryption schemes. We will review the Knuth-Yao random walk in detail in Chap. 4.

2.5.2 Polynomial arithmetic

The ring-LWE-based schemes in Sect. 2.4 perform polynomial arithmetic in $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$. Polynomial addition are subtraction can be performed in $\mathcal{O}(n)$ time simply by performing coefficient-wise additions or subtractions modulo q. Computation of polynomial multiplication is the costliest operation in the ring-LWE-based cryptographic schemes. In the general case, given two polynomials $a(x)$ and $b(x)$, first their product $c'(x) = a(x) \cdot b(x)$ is computed and then the product is reduced modulo $f(x)$ to get the polynomial multiplication result $c(x) = c'(x) \mod f(x)$. In the following we briefly review some of the well-known polynomial multiplication methods for computing $a(x) \cdot b(x)$. A survey of fast multiplication algorithms can be found in [16].

The school-book polynomial multiplication is the simplest method for computing the product of two polynomials. It computes the result as a summation of products using as follows.

$$a(x) \cdot b(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j x^{i+j} \ . \tag{2.9}$$

From the above equation it is clear that the school-book method has $\mathcal{O}(n^2)$ time complexity. The simplicity of this method makes it attractive for designing a compact polynomial multiplier for small n. For the ring-LWE-based schemes in Sect. 2.4, n is large and hence the school-book multiplication is not suitable.

Karatsuba polynomial multiplication uses a divide-and-conquer approach to perform polynomial multiplication. For n a power of two, the Karatsuba method divides the input polynomials into polynomials of half length. E.g. $a(x)$ is split into two polynomials a_h and a_l each having $n/2$ coefficients.

$$\begin{aligned} a_h &= a_{n-1}x^{n-1} + \ldots + a_{n/2}x^{n/2} \\ a_l &= a_{n/2-1}x^{n/2-1} + \ldots + a_0 \ . \end{aligned}$$

Similarly b_h and b_l are obtained after splitting $b(x)$. The multiplication is performed as follows.

$$a(x) \cdot b(x) = a_h \cdot b_h x^n + \big[(a_h + a_l) \cdot (b_h + b_l) - a_h \cdot b_h - a_l \cdot b_l\big] x^{n/2} + a_l \cdot b_l \ .$$

Thus the Karatsuba method has turned the n-coefficient polynomial multiplication into three $n/2$-coefficient polynomial multiplications. Following the same strategy, each of these smaller multiplications can be turned into even smaller multiplications. The Karatsuba method-based polynomial multiplication algorithms use recursive function calls to realize the divide-and-conquer strategy. The method has $\mathcal{O}(n^{\lg_2 3})$ time complexity.

The Fourier transform method is the most efficient method for computing polynomial multiplication. The n-point forward Discrete Fourier Transform (DFT) of a polynomial $a(x) = \sum_{k=0}^{n-1} a_k x^k$ consists of n evaluations of $a(x)$ at n distinct points. The good thing is that, polynomial multiplication in the Fourier domain turns into a coefficient-wise multiplication operation. Hence, if the input polynomials are provided in their Fourier representation, then we can multiply them in $\mathcal{O}(n)$ time. However the trivial way to compute the DFT has quadratic time complexity.

The Fast Fourier Transform (FFT) is an efficient way to compute the DFT of a polynomial in $O(n \lg n)$ time. It evaluates the input polynomial in the points ω_n^k for the integer $k \in [0, n-1]$, where ω_n is the n-th primitive root of the unity. The FFT applies a divide-and-conquer approach for the evaluation by exploiting a special property $\omega_n^{2k} = \omega_{n/2}^k$ of ω_n. For n a power of two, the input polynomial $a(x)$ is split into two smaller polynomials $a_e(x) = a^{n-2}x^{n/2-1} + \ldots + a_0$ and $a_o(x) = a^{n-1}x^{n/2-1} + \ldots + a_1$ such that $a(x) = a_e(x^2) + x a_o(x^2)$. The evaluations are performed only at the $n/2$ distinct points $\omega_{n/2}^k$ for $k \in [0, n/2-1]$. Now the evaluation of $a(x)$ is obtained by combining the evaluations of $a_e(x)$ and $a_o(x)$ as shown below and by using the special property of ω_n i.e. $\omega_n^{2k} = \omega_{n/2}^k$ for any $k > n/2$.

$$a(\omega_n^k) = a_e(\omega_n^{2k}) + \omega_n^k a_o(\omega_n^{2k}) \ .$$

In the FFT method, the n-th primitive root of unity ω_n is a complex number, and hence FFT involves floating point arithmetic. The Number Theoretic Transform (NTT) corresponds to an FFT where the roots of unity are taken from a finite ring $\mathbb{Z}_q$. Hence all computations in NTT are performed on integers. The NTT exists if and only if n divides $d-1$ for every prime divisor d of q. In this research we use the NTT to efficiently compute polynomial multiplication in R_q. In Chap. 5 we review an inplace iterative version of the NTT algorithm. For

two polynomials a and b, their multiplication using the NTT can be computed as follows.

$$a \cdot b = INTT_{\omega}^{2n}\left(NTT_{\omega}^{2n}(a) * NTT_{\omega}^{2n}(b)\right) .$$

Here $NTT_{\omega}^{2n}(\cdot)$ stands for a $2n$-point NTT, $INTT_{\omega}^{2n}(\cdot)$ stands for the inverse transform, and the operator $*$ stands for coefficient-wise multiplications.

Reduction modulo $f(x)$

In a polynomial multiplication over R_q, the result $c'(x) = a(x) \cdot b(x)$ which has $2n$ coefficients, is reduced modulo the irreducible polynomial $f(x)$. In the general case, i.e. when $f(x)$ does not possess any special structure, we can compute the modular reduction by computing the quotient $quo(x)$ and remainder $rem(x)$ polynomials such that $c'(x) = quo(x) \cdot f(x) + rem(x)$. The computation of $quo(x)$ requires a polynomial division operation. An efficient way to compute this division is to use the Newton iteration method. The steps are as follows [139].

Let $\texttt{rev}_k()$ be a function that reverses the positions of the first k coefficients of the input polynomial. For e.g., when $\texttt{rev}_3()$ is applied to the polynomial $3x^2 + 5x + 2$, the result is the polynomial $2x^2 + 5x + 3$.

From the equation $c' = quo \cdot f + rem$, where c' has degree $2n - 1$ and f has degree n, we get the relation

$$\texttt{rev}_{2n-1}(c') = \texttt{rev}_{n-1}(quo) \cdot \texttt{rev}_n(f) + x^n \texttt{rev}_{n-1}(rem) .$$

Therefore, we can get the congruence relation

$$\texttt{rev}_{n-1}(quo) \equiv \texttt{rev}_{2n-1}(c') \cdot \texttt{rev}_n(f)^{-1} \pmod{x^n} .$$

For arithmetic in R_q, the irreducible polynomial f is always constant and hence $\texttt{rev}_n(f)^{-1} \pmod{x^n}$ is a constant polynomial with n coefficients. We see that $\texttt{rev}_{n-1}(quo) \pmod{x^n}$ can be computed by performing simple polynomial multiplication and then taking the least n coefficients of the result. Computation of quo from $\texttt{rev}_{n-1}(quo)$ requires another application of $\texttt{rev}_{n-1}()$. Now the remainder is computed as $rem(x) = c'(x) - quo(x) \cdot f(x)$.

2.5.3 Division and rounding

Both FV.Mult and YASHE.Mult require divisions by q. The basic division algorithms [40] namely the restoring and non-restoring divisions are iterative in

nature and produce one digit of the quotient per iteration. Though accurate, such algorithms are very slow for homomorphic encryption schemes. Fast division methods use a reciprocal function computation to find a close approximation of the quotient. Such methods are also iterative but after each iteration the number of bits of accuracy of the approximate quotient doubles. As a result the number of iterations for fast division methods is less than for basic division methods. The Newton–Raphson and Goldschmidt methods are examples of fast division methods. A comprehensive study of fast division algorithms is provided in [41].

Note that the division operation that we need to perform always uses a constant value q as the divisor. When the divisor is fixed, we can do a lot better than the above mentioned methods as the reciprocal $1/q$ can be computed beforehand. The quotient can be computed by multiplying the dividend by the reciprocal. This multiplication can be computed efficiently if the underlying platform supports word level integer multiplications. To compute the rounding of the division result we use the following theorem.

Theorem 2. To round a division of two k-bit integers correctly to k-bits, the quotient must be computed correctly to $2k + 1$ bits [66].

Hence to round the result of dividing a k_1-bit dividend by a k_2-bit divisor we need to pre-compute the reciprocal upto $k_1 + k_2 + 1$ bits.

2.6 Summary

In this chapter, we revisited the elliptic-curve discrete logarithm problem and hard lattice problems, and introduced a set of PKC schemes based on these problems. For elliptic-curve cryptography over binary extension fields, we described the mathematics of the finite field primitives and introduced the Koblitz curves for faster point multiplication. For lattice-based cryptography over polynomial rings, we briefly introduced the mathematics of polynomial arithmetic and discrete Gaussian sampling, and described the well-known methods for implementing these primitives.

In the next chapter we will design a lightweight Koblitz curve point multiplier and we will perform several lightweight optimizations to implement the scalar conversion. Then, from Chap. 4 onwards we will optimize the primitives for the ring-LWE-based PKC and design hardware architectures for the PKC schemes.

Chapter 3

Coprocessor for Koblitz curves

CONTENT SOURCES:

Sujoy Sinha Roy, Kimmo Järvinen, Ingrid Verbauwhede Lightweight Coprocessor for Koblitz Curves: 283-Bit ECC Including Scalar Conversion with only 4300 Gates In *Cryptographic Hardware and Embedded Systems CHES* (2015).

Contribution: Main author.

Sujoy Sinha Roy, Junfeng Fan, Ingrid Verbauwhede Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2015).

Contribution: Main author.

3.1 Introduction

Koblitz curves [70] are a special class of elliptic-curves which enable very efficient point multiplications and, therefore, they are attractive for hardware and software implementations. However, these efficiency gains can be exploited only by representing scalars as specific τ-adic expansions. Most cryptosystems require the scalar also as an integer (see, e.g., ECDSA [94]). Therefore, cryptosystems

utilizing Koblitz curves need both the integer and τ-adic representations of the scalar, which results in a need for conversions between the two domains.

In the literature a few research works exist that address the challenges of designing the scalar conversion operation in hardware. The first conversion architecture was proposed in [72] and was followed by [135, 140]. Still the converters were slow and had large area requirements. Brumley and Järvinen in [23] proposed an efficient conversion algorithm tailored for hardware platforms. Due to its sequential nature, the authors named the algorithm *lazy reduction*. The algorithm uses only integer addition, subtraction and shifting operations; no multi-precision divisions or multiplications are used. Later a speed optimized version of the lazy reduction, known as the *double lazy reduction* was proposed in [62] by Adikari, Dimitrov, and Järvinen. Still, the extra overhead introduced by these conversions has so far prevented efforts to use Koblitz curves in lightweight implementations. Lightweight applications that require elliptic-curve cryptography include, e.g., wireless sensor network nodes, RFID tags, medical implants, and smart cards. Such applications will have a central role in actualizing concepts such as the Internet of Things. However such applications have strict constraints on implementation resources such as power, energy, circuit area, memory, etc. Since the Koblitz curves are a class of computationally efficient elliptic-curves, they could potentially be the right choice for the lightweight applications. Indeed [11] showed that Koblitz curves result in a very efficient lightweight implementation if τ-adic expansions are already available. But the fact that the conversion is not included seriously limits possible applications of the implementation.

In this chapter we investigate a design methodology for implementing high security Koblitz curve cryptoprocessor on hardware platforms. We choose the NIST [92] recommended 283 bit Koblitz curve which offers around 140 bit security. We introduce several optimizations in the conversion of integer scalars. Finally, we design a lightweight coprocessor architecture of the 283-bit Koblitz curve point multiplier by using the lightweight scalar conversion architecture. We also include a set of countermeasures against timing attacks, simple power analysis (SPA), differential power analysis (DPA) and safe-error fault attacks.

The remaining part of the chapter is organized as follows: In Sect. 3.2 we optimize the Koblitz scalar conversion operation and introduce several lightweight countermeasures against side channel attacks. In Sect. 3.3 we describe the point multiplication operation. We use these optimization techniques to design a lightweight coprocessor architecture in Sect. 3.4. We provide synthesis results in 130 nm CMOS and comparisons to other works in Sect. 3.5. The final section summarizes the contributions.

Algorithm 2: Scalar reduction algorithm from [23]

Input: Integer scalar k

Output: Reduced scalar $\rho = b_0 + b_1\tau \equiv k \pmod{\tau^m - 1}$

1 $(a_0, a_1) \leftarrow (1, 0)$, $(b_0, b_1) \leftarrow (0, 0)$, $(d_0, d_1) \leftarrow (k, 0)$;
2 **for** $i = 0$ **to** $m - 1$ **do**
3 $\quad u \leftarrow d_0[0]$; `/* lsb of` d_0 `is the remainder before division */`
4 $\quad d_0 \leftarrow d_0 - u$;
5 $\quad (b_0, b_1) \leftarrow (b_0 + u \cdot a_0, b_1 + u \cdot a_1)$;
6 $\quad (d_0, d_1) \leftarrow (d_1 - d_0/2, -d_0/2)$; `/* Division of` (d_0, d_1) `by` τ `*/`
7 $\quad (a_0, a_1) \leftarrow (-2a_1, a_0 - a_1)$;
8 **end**
9 $\rho = (b_0, b_1) \leftarrow (b_0 + d_0, b_1 + d_1)$;

3.2 Koblitz curve scalar conversion

As described in Sect. 2.2.1, the scalar conversion is performed in two phases: first the integer scalar k is reduced to $\rho = b_0 + b_1\tau \equiv k \pmod{\tau^m - 1}$ and then the τ-adic representation t is generated from the reduced scalar ρ [129, 97, 141]. The overhead of these conversions is specifically important for efficient implementations. Another important aspect is resistance against side-channel attacks. Only SPA countermeasures are required because only one conversion is required per k. The scalar k is typically a nonce but even if it is used multiple times, t can be computed only once and stored.

3.2.1 Scalar reduction

We choose the scalar reduction technique called lazy reduction (described as Alg. 2) from [23]. The scalar k is repeatedly divided by τ for m number of times to get the following relation.

$$\begin{aligned} k &= (d_0 + d_1\tau)\tau^m + (b_0 + b_1\tau) \\ &= (d_0 + d_1\tau)(\tau^m - 1) + (b_0 + d_0) + (b_1 + d_1)\tau \\ &= \lambda(\tau^m - 1) + \rho \ . \end{aligned}$$

As shown in Theorem 1 in Sect. 2.2.1, this division can be implemented with shifts, additions, and subtractions. This makes the scalar reduction algorithm attractive for lightweight implementations. The τ-adic representation generated from ρ has a length at most $m + 4$ in F_{2^m} [23].

To meet the constraints of a lightweight platform, we implement the lazy reduction algorithm [23] in a word-serial fashion. Though this design decision reduces the area requirement, it increases the cycle count. Hence we optimize further the computational steps of Alg. 2 to reduce the number of cycles. Further, we investigate side-channel vulnerability of the algorithm and propose lightweight countermeasures against SPA.

Computational optimization

In lines 6 and 7 of Alg. 2, computations of d_1 and a_0 require subtractions from zero. In a word-serial architecture with only one adder/subtracter circuit, they consume nearly 33% of the cycles of the scalar reduction. We use the iterative property of Alg. 2 and eliminate these two subtractions by replacing lines 6 and 7 with the following ones:

$$\begin{aligned}(d_0, d_1) &\leftarrow (d_0/2 - d_1, d_0/2)\\ (a_0, a_1) &\leftarrow (2a_1, a_1 - a_0)\ .\end{aligned} \tag{3.1}$$

However with this modification, (a_0, a_1) and (d_0, d_1) have a wrong sign after every odd number of iterations of the for-loop in Alg. 2. It may appear that this wrong sign could affect correctness of (b_0, b_1) in line 5. Since the remainder u (in line 3) is generated from d_0 instead of the correct value $-d_0$, a wrong sign is also assigned to u. Hence, the multiplications $u \cdot a_0$ and $u \cdot a_1$ in line 5 are always correct, and the computation of (b_0, b_1) remains unaffected of the wrong signs.

After completion of the for-loop, the sign of (d_0, d_1) is wrong as m is an odd integer for secure fields. Hence, the correct value of the reduced scalar should be computed as $\rho \leftarrow (b_0 - d_0, b_1 - d_1)$.

Protection against SPA

In line 5 of Alg. 2, computation of new (b_0, b_1) depends on the remainder bit (u) generated from d_0 which is initialized to k. Multi-precision additions are performed when $u = 1$; whereas no addition is required when u is zero. A side-channel attacker can detect this conditional computation and can use, e.g., the techniques from [23] to reconstruct the secret key from the remainder bits that are generated during the scalar reduction.

One way to protect the scalar reduction from SPA is to perform dummy additions $(b'_0, b'_1) \leftarrow (b_0 + a_0, b_1 + a_1)$ whenever $u = 0$. However, such countermeasures

Algorithm 3: SPA-resistant scalar reduction

Input: Integer scalar k

Output: Reduced scalar $\rho = b_0 + b_1\tau \equiv k \pmod{\tau^m - 1}$

$(a_0, a_1) \leftarrow (1, 0)$, $(b_0, b_1) \leftarrow (0, 0)$, $(d_0, d_1) \leftarrow (k, 0)$;
if $d_0[0] = 0$ **then**
  $e \leftarrow 1$; /* Set to 1 when d_0 is even */
  $d_0[0] \leftarrow 1$;
end
for $i = 0$ **to** $m - 1$ **do**
  $u \leftarrow \Psi(d_0 + d_1\tau)$; /* Remainder $u \in \{1, -1\}$, computed using (3.2) */
  $d_0 \leftarrow d_0 - u$;
  $(b_0, b_1) \leftarrow (b_0 + u \cdot a_0, b_1 + u \cdot a_1)$;
  $(d_0, d_1) \leftarrow (d_0/2 - d_1, d_0/2)$; /* Saves one subtraction */
  $(a_0, a_1) \leftarrow (2a_1, a_1 - a_0)$; /* Saves one subtraction */
end
$\rho = (b_0, b_1) \leftarrow (b_0 - d_0 - e, b_1 - d_1)$; /* Subtraction */

based on dummy operations require more memory and are vulnerable to C safe-error fault attacks [42]. We propose a countermeasure inspired by the zero-free τ-adic representations from [97, 141]. A zero-free representation is obtained by generating the remainders u from $d = d_0 + d_1\tau$ using a map $\Psi(d) \rightarrow u \in \{1, -1\}$ such that $d - u$ is divisible by τ, but additionally not divisible by τ^2 (see Sect. 3.2.2). We observe that during the scalar reduction (which is basically a division by τ), we can generate the remainder bits u as either 1 or -1 throughout the entire for-loop in Alg. 2. Because $u \neq 0$, a new (b_0, b_1) is always computed in the for-loop and protection against SPA is achieved without dummy operations. The following equation generates u by observing the second lsb of d_0 and lsb of d_1.

$$\begin{aligned} &\text{Case 1: If } d_0[1] = 0 \text{ and } d_1[0] = 0, \text{ then } u \leftarrow -1 \\ &\text{Case 2: If } d_0[1] = 1 \text{ and } d_1[0] = 0, \text{ then } u \leftarrow 1 \\ &\text{Case 3: If } d_0[1] = 0 \text{ and } d_1[0] = 1, \text{ then } u \leftarrow 1 \\ &\text{Case 4: If } d_0[1] = 1 \text{ and } d_1[0] = 1, \text{ then } u \leftarrow -1 \ . \end{aligned} \tag{3.2}$$

The above equation takes an odd d_0 and computes u such that the new d_0 after division of $d - u$ by τ is also an odd integer.

Alg. 3 shows our computationally efficient SPA-resistant scalar reduction algorithm. All operations are performed in a word-serial fashion. Since the remainder generation in (3.2) requires the input d_0 to be an odd integer, the lsb

Algorithm 4: Computation of zero-free τ-adic representation [97]

Input: Reduced scalar $\rho = b_0 + b_1\tau$ with b_0 odd
Output: Zero-free τ-adic bits $(t_{\ell-1}, \cdots t_0)$

1 $i \leftarrow 0$;
2 **while** $|b_0| \neq 1$ **or** $b_1 \neq 0$ **do**
3 $\quad u \leftarrow \Psi(b_0 + b_1\tau)$; `/* Computed using (3.2) */`
4 $\quad b_0 \leftarrow b_0 - u$;
5 $\quad (b_0, b_1) \leftarrow (b_1 - b_0/2, -b_0/2)$;
6 $\quad t_i \leftarrow u$;
7 $\quad i \leftarrow i + 1$;
8 **end**
9 $t_i \leftarrow b_0$;

of d_0 is always set to 1 (in line 4) when the input scalar k is an even integer. In this case, the algorithm computes the reduced scalar of $k + 1$ instead of k and after the completion of the reduction, the reduced scalar should be decremented by one. Alg. 3 uses a one-bit register e to implement this requirement. The final subtraction in line 13 uses e as a borrow to the adder/subtracter circuit. In the next section, we show that the subtraction $d_0 - u$ in line 8 also leaks information about u and propose a countermeasure that prevents this.

3.2.2 Computation of τ-adic representation

For side-channel attack resistant point multiplication, we use the zero-free τ-adic representation proposed in [97, 141] and described in Alg. 4. We add the following improvements to the algorithm.

Computational optimization

Computation of b_1 in line 5 of Alg. 4 requires subtraction from zero. Similar to Sect. 3.2.1 this subtraction can be avoided by computing $(b_0, b_1) \leftarrow (b_0/2 - b_1, b_0/2)$. With this modification, the sign of (b_0, b_1) will be wrong after an odd number of iterations. In order to correct this, the sign of t_i should be flipped for odd i (by multiplying it with $(-1)^i$).

Protection against SPA

Though point multiplications with zero-free representations are resistant against SPA [97], the generation of τ-adic bits (Alg. 4) is vulnerable to SPA. In line 3 of Alg. 4, a remainder u is computed as per the four different cases described in (3.2) and then subtracted from b_0 in line 4. We use the following observations to detect the side-channel vulnerability in this subtraction and to propose a countermeasure against SPA.

1. For Case 1, 2 and 3 in (3.2), the subtractions of u are equivalent to flipping two (or one) least significant bits of b_0. Hence, actual subtractions are not computed in these cases.
2. For Case 4, subtraction of u from b_0 (i.e. computation of $b_0 + 1$) involves carry propagation. Hence, an actual multi-precision subtraction is computed in this case.
3. If any iteration of the while-loop in Alg. 4 meets Case 4, then the new value of b_1 will be even. Hence, the while-loop will meet either Case 1 or Case 2 in the next iteration.

Based on the differences in computation, a side-channel attacker using SPA can distinguish Case 4 from the other three cases. Hence, the attacker can reveal around 25% of the bits of a zero-free representation. Moreover, the attacker knows that the following τ-adic bits are biased towards 1 instead of -1 with a probability of 1/3.

We propose a very low-cost countermeasure that skips this special addition $b_0 + 1$ for Case 4 by merging it with the computation of the new (b_0, b_1) in Alg. 4. In line 5, we compute a new b_0 as:

$$b_0 \leftarrow \left(\frac{b_0 + 1}{2} - b_1\right) = \left(\frac{b_0 - 1}{2} - \{b'_1, 0\}\right) \quad . \tag{3.3}$$

Since b_1 is an odd number for Case 4, we can represent it as $\{b'_1, 1\}$ and subtract the least significant bit 1 from $(b_0 + 1)/2$ to get $(b_0 - 1)/2$. Since b_0 is always odd, the computation of $(b_0 - 1)/2$ is just a left-shift of b_0.

The computation of $b_1 \leftarrow (b_0+1)/2$ in line 5 of Alg. 4 involves a carry propagation and thus an actual addition becomes necessary. We solve this problem by computing $b_1 \leftarrow (b_0 - 1)/2$ instead of the correct value $b_1 \leftarrow (b_0 + 1)/2$ and remembering the difference (i.e., 1) in a flag register h. Correctness of the τ-adic representation can be maintained by considering this difference in the

future computations that use this wrong value of b_1. Now as per observation 3, the next iteration of the while-loop meets either Case 1 or 2. We adjust the previous difference by computing the new b_0 as follows:

$$b_0 \leftarrow \left(\frac{b_0}{2} - (b_1 + h)\right) = \left(\frac{b_0}{2} - b_1 - 1\right) \quad . \tag{3.4}$$

In a hardware architecture, this equation can be computed by setting the borrow input of the adder/subtracter circuit to 1 during the subtraction.

In (3.5), we show our new map $\Psi'(\cdot)$ that computes a remainder u and a new value h' of the difference flag following the above procedure. We consider $b_1[0] \oplus h$ (instead of $b_1[0]$ as in (3.2)) because a wrong b_1 is computed in Case 4 and the difference is kept in h.

$$\begin{aligned}
&\text{Case 1: If } b_0[1] = 0 \text{ and } b_1[0] \oplus h = 0, \text{ then } u \leftarrow -1 \text{ and } h' \leftarrow 0 \\
&\text{Case 2: If } b_0[1] = 1 \text{ and } b_1[0] \oplus h = 0, \text{ then } u \leftarrow 1 \quad \text{and } h' \leftarrow 0 \\
&\text{Case 3: If } b_0[1] = 0 \text{ and } b_1[0] \oplus h = 1, \text{ then } u \leftarrow 1 \quad \text{and } h' \leftarrow 0 \\
&\text{Case 4: If } b_0[1] = 1 \text{ and } b_1[0] \oplus h = 1, \text{ then } u \leftarrow -1 \text{ and } h' \leftarrow 1 \ .
\end{aligned} \tag{3.5}$$

The same technique is also applied to protect the subtraction $d_0 - u$ in the scalar reduction in Alg. 3.

Protection against timing attack

The terminal condition of the while-loop in Alg. 4 is dependent on the input scalar. Thus by observing the timing of the computation, an attacker is able to know the higher order bits of a short τ-adic representation. This allows the attacker to narrow down the search domain. We observe that we can continue the generation of zero-free τ-adic bits even when the terminal condition in Alg. 4 is reached. In this case, the redundant part of the τ-adic representation is equivalent to the value of b_0 when the terminal condition was reached for the first time; hence the result of the point multiplication remains correct. For example, starting from $(b_0, b_1) = (1, 0)$, the algorithm generates an intermediate zero-free representation $-\tau - 1$ and again reaches the terminal condition $(b_0, b_1) = (-1, 0)$. The redundant representation $-\tau^2 - \tau - 1$ is equivalent to 1. If we continue, then the next terminal condition is again reached after generating another two bits. In this chapter we generate zero-free τ-adic representations that have lengths always larger than or equal to m of the field $\mathbb{F}_{2^m}$. To implement this feature, we added the terminal condition $i < m$ to the while-loop.

Algorithm 5: SPA-resistant generation of a zero-free τ-adic representation

Input: Reduced scalar $\rho = b_0 + b_1\tau$
Output: τ-adic bits $(t_{ell-1}, \cdots t_0)$ and flag f

1 $f \leftarrow$ assign_flag$(b_0[0], b_1[0])$;
2 $(b_0[0], b_1[0]) \leftarrow$ bitflip$(b_0[0], b_1[0], f)$; `/* Initial adjustment */`
3 $i \leftarrow 0$;
4 $h \leftarrow 0$;
5 **while** $i < m$ **or** $|b_0| \neq 1$ **or** $(b_1 \neq 0$ **and** $h = 0)$ **or** $(b_1 \neq -1$ **and** $h = 1)$ **do**
6 $\quad (u, h') \leftarrow \Psi'(b_0 + b_1\tau)$; `/* Computed using (3.5) */`
7 $\quad b_0[1] \leftarrow \neg b_0[1]$; `/* Second LSB is set to 1 when Case 1 occurs */`
8 $\quad (b_0, b_1) \leftarrow (\frac{b_0}{2} - b_1 - h, \frac{b_0}{2})$;
9 $\quad t_i \leftarrow (-1)^i \cdot u$;
10 $\quad h \leftarrow h'$;
11 $\quad i \leftarrow i + 1$;
12 **end**
13 $t_i \leftarrow (-1)^i \cdot b_0$;

In Alg. 5, we describe an algorithm for generating zero-free representations that applies the proposed computational optimizations and countermeasures against SPA and timing attacks. The while-loops of both Alg. 4 and 5 require b_0 to be an odd integer. When the input ρ has an even b_0, then an adjustment is made by adding one to b_0 and adding (subtracting) one to (from) b_1 when b_1 is even (odd). This adjustment is recorded in a flag f in the following way: if b_0 is odd, then $f = 0$; otherwise $f = 1$ or $f = 2$ depending on whether b_1 is even or odd, respectively. In the end of a point multiplication, this flag is checked and $(\tau + 1)P$ or $(-\tau + 1)P$ is subtracted from the point multiplication result if $f = 1$ or $f = 2$, respectively. This compensates the initial addition of $(\tau + 1)$ or $(-\tau + 1)$ to the reduced scalar ρ described in line 2 of Alg. 5.

We also designed a high-speed scalar conversion architecture based on [62]. The optimization strategy and the hardware architecture is described in the appendix A.

3.3 Point multiplication

We base the point multiplication algorithm on the use of the zero-free representation discussed in Sect. 3.2. We give our modification of the point multiplication algorithm of [97, 141] with window size $w = 2$ in Alg. 6. The algorithm includes countermeasures against SPA, DPA, and timing attacks as

well as inherent resistance against C safe-error fault attacks. Implementation details of each operation used by Alg. 6 are given in App. B. Below, we give a high-level description.

Algorithm 6: Zero-free point multiplication with side-channel countermeasures

Input: An integer k, the base point $P = (x, y)$, a random element $r \in \mathbb{F}_{2^m}$
Output: The result point $Q = kP$

1 $(t, f) \leftarrow \text{Convert}(k)$; /* Alg. 3 and 5 */
2 $P_{+1} \leftarrow \phi(P) + P$;
3 $P_{-1} \leftarrow \phi(P) - P$;
4 **if** ℓ *is odd* **then** $Q = (X, Y) \leftarrow t_{\ell-1}P$; $i \leftarrow \ell - 3$;
5 ;
6 **else** $Q = (X, Y) \leftarrow t_{\ell-1}P_{t_{\ell-2}t_{\ell-1}}$; $i \leftarrow \ell - 4$;
7 ;
8 $Q = (X, Y, Z) \leftarrow (Xr, Yr^2, r)$;
9 **while** $i \geq 0$ **do**
10 $\quad Q \leftarrow \phi^2(Q)$;
11 $\quad Q \leftarrow Q + t_{i+1}P_{t_i t_{i+1}}$;
12 $\quad i \leftarrow i - 2$;
13 **end**
14 **if** $f = 1$ **then** $Q \leftarrow Q + P_{-1}$;
15 ;
16 **else if** $f = 2$ **then** $Q \leftarrow Q - P_{+1}$;
17 ;
18 $Q = (X, Y) \leftarrow (X/Z, Y/Z^2)$;
19 **return** Q ;

Line 1 computes the zero-free representation t given an integer k using Alg. 3 and 5. It outputs a zero-free expansion of length ℓ with $t_i \in \{-1, +1\}$ represented as an ℓ-bit vector and a flag f. Lines 2 and 3 perform the precomputations by computing $P_{+1} = \phi(P) + P$ and $P_{-1} = \phi(P) - P$. Lines 4 and 5 initialize the accumulator point Q depending on the length of the zero-free expansion. If the length is odd, then Q is set to $\pm P$ depending on the msb $t_{\ell-1}$. If the length is even, then Q is initialized with $\pm\phi(P) \pm P$ by using the precomputed points depending on the values of the two msb's $t_{\ell-1}$ and $t_{\ell-2}$. Line 6 randomizes Q by using a random element $r \in \mathbb{F}_{2^m}$ as suggested by Coron [29]. This randomization offers protection against DPA and attacks that calculate hypotheses about the values of Q based on its known initial value (e.g., the doubling attack [44]).

Lines 7 to 10 iterate the main loop of the algorithm by observing two bits of the zero-free expansion on each iteration. Each iteration begins in line 8 by computing two Frobenius endomorphisms. Line 9 either adds or subtracts

$P_{+1} = (x_{+1}, y_{+1})$ or $P_{-1} = (x_{-1}, y_{-1})$ to or from Q depending on the values of t_i and t_{i+1} processed by the iteration. It is implemented by using the equations from [5] which compute a point addition in mixed affine and López-Dahab [82] coordinates. Point addition and subtraction are carried out with the exactly same pattern of operations (see App. B). Lines 11 and 12 correct the adjustments that ensure that b_0 is odd before starting the generation of the zero-free representation (see Sect. 3.2.2). Line 13 retrieves the affine point of the result point Q.

The pattern of operations in Alg. 6 is almost constant. The side-channel properties of the conversion (line 1) were discussed in Sect. 3.2. The precomputation (lines 2 and 3) is fixed and operates only on the base point, which is typically public. The initialization of Q (lines 4 and 5) can be carried out with a constant pattern of operations with the help of dummy operations. The randomization of Q protects from differential power analysis (DPA) and comparative side-channel attacks (e.g., the doubling attack [44]). The main loop operates with a fixed pattern of operations on a randomized Q offering protecting against SPA and DPA. Lines 11 and 12 depend on t (and, thus, k) but they leak at most one bit to an adversary who can determine whether they were computed or not. This leakage can be prevented with a dummy operation. Although the algorithm includes dummy operations, it offers good protection also against C safe-error fault attacks. The reason is that the main loop does not involve any dummy operations and, hence, even an attacker, who is able to distinguish dummy operations, learns only few bits of information (at most, the lsb and the msb and whether the length is odd or even). Hence, C safe-error fault attacks that aim to reveal secret information by distinguishing dummy operations are not a viable attack strategy [42].

3.4 Architecture

In this section, we describe the hardware architecture (Fig. 3.1) of our ECC coprocessor for 16-bit microcontrollers such as TI MSP430F241x or MSP430F261x [133]. Such families of low-power microcontrollers have at least 4KB of RAM and can run at 16 MHz clock. We connect our coprocessor to the microcontroller using a memory-mapped interface [124] following the drop-in concept from [144] where the coprocessor is placed on the bus between the microcontroller and the RAM and memory access is controlled with multiplexers. The coprocessor consists of the following components: an arithmetic and logic unit (ALU), an address generation unit, a shared memory and a control unit composed of hierarchical finite state machines (FSMs).

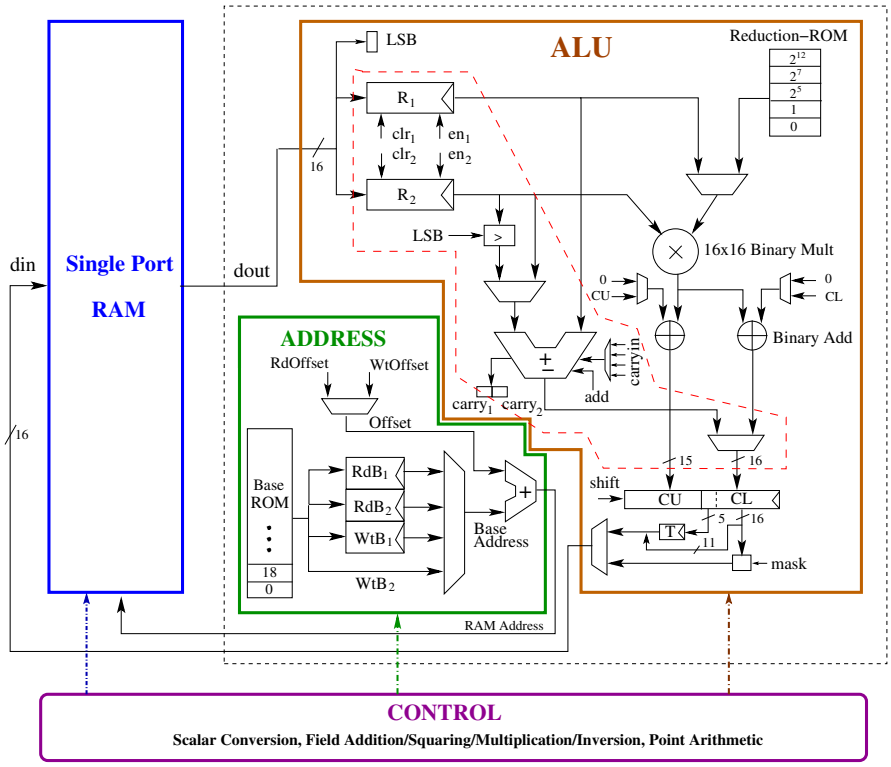

Figure 3.1: Hardware architecture of the ECC coprocessor

The arithmetic and logic unit (ECC-ALU)

has a 16-bit data path and is used for both integer and binary field computations. The ECC-ALU is interfaced with the memory block using an input register pair (R_1, R_2) and an output multiplexer. The central part of the ECC-ALU consists of a 16-bit integer adder/subtracter circuit, a 16-bit binary multiplier and two binary adders. A small *Reduction-ROM* contains several constants that are used during modular reductions and multiplications by constants. The accumulator register pair (CU, CL) stores the intermediate or final results of any arithmetic operation. Finally, the output multiplexer is used to store the contents of the registers CL, T and a masked version of CL in the memory block, which sets the msb's of the most significant word of an alement to zero.

The memory block

is a single-port RAM which is shared by the ECC coprocessor and the 16-bit microcontroller. Each 283-bit element of $\mathbb{F}_{2^{283}}$ requires 18 16-bit words totaling 288 bits. The coprocessor requires storage for 14 elements of $\mathbb{F}_{2^{283}}$ (see App. B), which gives 4032 bits of RAM (252 16-bit words). Some of these variables are reused for different purposes during the conversion.

The address unit

generates address signals for the memory block. A small *Base-ROM* is used to keep the base addresses for storing different field elements in the memory. During any integer operation or binary field operation, the two address registers RdB_1 and RdB_2 in the address unit are loaded with the base addresses of the input operands. Similarly the base addresses for writing intermediate or final results in the memory block are provided in the register WtB_1 and in the output from the *Base-ROM* (WtB_2). The adder circuit of the address block is an 8-bit adder which computes the physical address from a read/write offset value and a base address.

The control unit

consists of a set of hierarchical FSMs that generate control signals for the blocks described above. The FSMs are described below.

1) Scalar Conversion uses the part of the ECC-ALU shown by the red dashed polygon in Fig. 3.1. The computations controlled by this FSM are mainly integer additions, subtractions and shifts. During any addition or subtraction, the words of the operands are first loaded in the register pair (R_1, R_2). The result-word is computed using the integer adder/subtracter circuit and stored in the accumulator register CL. During a right-shift, R_2 is loaded with the operand-word and R_1 is cleared. Then the lsb of the next higher word of the operand is stored in the one-bit register *LSB*. Now the integer adder is used to add the shifted value $\{LSB, R_2/2\}$ with R_1 to get the shifted word. One scalar conversion requires around 78,000 cycles.

2) Binary Field Primitives use the registers and the portion of the ECC-ALU outside the red-dashed polygon in Fig. 3.1.

- Field addition sequentially loads two words of the operands in R_2, then multiplies the words by 1 (from the *Reduction-ROM*) and finally calculates the result-word in CL after accumulation. One field addition requires 60 cycles.
- Field multiplication uses word-serial comb method [54]. It loads the words of the operands in R_1 and R_2, then multiplies the words and finally accumulates. After the completion of the comb multiplication, a modular reduction is performed requiring mainly left-shifts and additions. The left-shifts are performed by multiplying the words with the values from the *Reduction-ROM*. One field multiplication requires 829 cycles.
- Field squaring computes the square of an element of $\mathbb{F}_{2^{283}}$ in linear time by squaring its words. The FSM first loads a word in both R_1 and R_2 and then squares the word by using the binary multiplier. After squaring the words, the FSM performs a modular reduction. The modular reduction is shared with the field multiplication FSM. One field squaring requires 200 cycles.
- Field inversion uses the Itoh-Tsujii algorithm [61] and performs field multiplications and squarings following an addition chain $(1, 2, 4, 8, 16, 17, 34, 35, 70, 140, 141, 282)$ for $\mathbb{F}_{2^{283}}$. One inversion requires 65,241 cycles.

3) Point Operations and Point Multiplication are implemented by combining an FSM with a hardwired program ROM. The program ROM includes subprograms for all operations of Alg. 6 and the address of the ROM is controlled by the FSM in order to execute Alg. 6 (see App. B for details).

Alg. 6 is executed so that the microcontroller initializes the addresses reserved for the accumulator point Q with the base point (x, y) and the random element r by writing $(X, Y, Z) \leftarrow (x, y, r)$. The scalar k is written into the RAM before the microcontroller issues a start point multiplication command. When this command is received, the reduction part of the conversion is executed followed by the computation of the msb(s) of the zero-free expansion. After this, the precomputations are performed by using (x, y) and the results are stored into the RAM. The initialization of Q is performed by writing either P_{+1} or P_{-1} in (X, Y) if the length of the expansion is even; otherwise, a dummy write is performed. Similarly, the sign of Q is changed if $t_{\ell-1} = -1$ and a dummy operation is computed otherwise. The main loop first executes two Frobenius endomorphisms and, then, issues an instruction that computes the next two bits of the zero-free expansion. By using these bits, either a point addition or a point subtraction is computed with P_{+1} or P_{-1}. One iteration of the main loop takes 9537 clock cycles. In the end, the affine coordinates of the result point

are retrieved and they become available for the microcontroller in the addresses for the X and Y coordinates of Q.

3.5 Results and comparisons

We described the architecture of Sect. 3.4 by using mixed Verilog and VHDL and simulated it with ModelSim SE 6.6d. We synthesized the code with Synopsys Design Compiler D-2010.03-SP4 using the regular compile for UMC 130 nm CMOS with voltage of 1.2 V by using Faraday FSC0L low-leakage standard cell libraries. The area given by the synthesis is 4,323 GE including everything in Fig. 3.1 except the single-port RAM. Computing one point multiplication requires in total 1,566,000 clock cycles including the scalar conversion. The power consumption at 16 MHz is 97.70 μW which gives an energy consumption of approximately 9.56 μJ per point multiplication. Table 3.1 summarizes our synthesis results together with several other lightweight ECC implementations from the literature. Since several of the reported implementations do not provide details of the libraries they used, we mention only the CMOS technology in the table. For a similar technology, there will be small variations in the area, power and energy consumption for different libraries.

Among all lightweight ECC processors available in the literature, the processor from [11] is the closest counterpart to our implementation because it is so far the only one that uses Koblitz curves. Even it has many differences with our architecture which make fair comparison difficult. The most obvious difference is that the processor from [11] is designed for a less secure Koblitz curve NIST K-163. Also the architecture of [11] differs from ours in many fundamental ways: they use a finite field over normal basis instead of polynomial basis, they use a bit-serial multiplier that requires all bits of both operands to be present during the entire multiplication instead of a word-serial architecture that we use, they store all variables in registers embedded into the processor architecture instead of an external RAM, and they also do not provide support for scalar conversions or any countermeasures against side-channel attacks. They also provide implementation results on 65 nm CMOS. Our architecture is significantly more scalable for different Koblitz curves because, besides control logic and RAM requirements, other parts remain almost the same, whereas the entire multiplier needs to be changed for [11]. It is also hard to see how scalar conversions or side-channel countermeasures could be integrated into the architecture of [11] without significant increases on both area and latency.

Table 3.1 includes also implementations that use the binary curve B-163 and the prime curve P-160 from [94]. The area of our coprocessor is on the level

of the smallest coprocessors available in the literature. Hence, the effect of selecting a 283-bit elliptic-curve instead of a less secure curve is negligible in terms of area. The price to pay for higher security comes in the form of memory requirements and computation latency. The amount of memory is not a major issue because our processor shares the memory with the microcontroller which typically has a large memory (e.g. TI MSP430F241x and MSP430F261x have at least 4KB RAM [133]). Also the computation time is on the same level with other published implementations because our coprocessor is designed to run on the relatively high clock frequency of the microcontroller which is 16 MHz.

In this work our main focus was to investigate feasibility of lightweight implementations of Koblitz curves for applications demanding high security. To enable a somewhat fair comparison with the existing lightweight implementations over $\mathbb{F}_{2^{163}}$, Table 3.1 provides estimates for area and cycles of ECC coprocessors that follow the design decisions presented in this chapter and perform point multiplications on curves B-163 or K-163. Our estimated cycle count for scalar multiplication over $\mathbb{F}_{2^{163}}$ is based on the following facts:

1. A field element in $\mathbb{F}_{2^{163}}$ requires 11 16-bit words, and hence, is smaller by a factor of 0.61 than a field element in $\mathbb{F}_{2^{283}}$. Since field addition and squaring have linear complexity, we estimate that the cycle counts for these operations scale down by a factor of around 0.61 and become 37 and 122 respectively. In a similarly way we estimate that field multiplication (which has quadratic complexity) scales down to 309 cycles. A field inversion operation following an addition chain $(1, 2, 4, 5, 10, 20, 40, 81, 162)$ requires nearly 22,700 cycles.

2. The for-loop in the scalar reduction operation (Alg. 3) executes 163 times in $\mathbb{F}_{2^{163}}$ and performs linear operations such as additions/subtractions and shifting. Moreover the length of τ-adic representation of a scalar reduces to 163 (thus reducing by a factor of 0.57 in comparison to $\mathbb{F}_{2^{283}}$). So, we estimate that the cycle count for scalar conversion scales down by a factor of 0.57×0.61 and requires nearly 27,000 cycles.

3. One Frobenius-and-add operation over $\mathbb{F}_{2^{283}}$ in Alg. 6 spends total 9,537 cycles among which 6,632 cycles are spent in eight quadratic-time field multiplications, and the rest 2,905 cycles are spent in linear-time operations. After scaling down, the cycle count for one Frobenius-and-add operation over $\mathbb{F}_{2^{163}}$ can be estimated to be around 4,250. The point multiplication loop iterates nearly 82 times for a τ-adic representation of length 164. Hence the number of cycles spent in this loop can be estimated to be around 348,500.

4. The precomputation and the final conversion steps are mainly dominated by the cost of field inversions. Hence the cycle counts can be estimated to be around 45,400.

As per the above estimates we see that a point multiplication using K-163 requires nearly 420,900 cycles. Similarly, we estimate that Montgomery's ladder for B-163 requires nearly 485,000 cycles. Our estimates show that our coprocessors for both B-163 and K-163 require more cycles in comparison to [145] which also uses a 16-bit ALU. The reason behind this is that [145] uses a dual-port RAM, whereas our implementation uses a single-port RAM (as it works as a coprocessor of MSP430). Moreover [145] has a dedicated squarer circuit to minimize cycle requirement for squaring.

Table 3.1 provides estimates for cycle and area of a modified version of the coprocessor that performs point multiplications using the Montgomery's ladder on the NIST curve B-283. The estimated cycle count is calculated from the cycle counts of the field operations described in Sect. 3.4. From the estimated value, we see that a point multiplication on B-283 requires nearly 23.5% more time. However, the coprocessor for B-283 is smaller by around 550 GE as no scalar conversion is needed.

Although application-specific integrated circuits are the primary targets for our coprocessor, it may be useful also for FPGA-based implementations whenever small ECC designs are needed. Hence, we compiled our coprocessor also for Xilinx Spartan-6 XC6SLX4-2TQG144 FPGA by using Xilinx ISE 13.4 Design Suite. After place and route, it requires only 209 slices (634 LUTs and 309 registers) and runs on clock frequencies up to 106.598 MHz.

Our coprocessor significantly improves speed, both classical and side-channel security, memory, and energy consumption compared to leading lightweight software [9, 32, 56, 64, 131]. For example, [32] reports a highly optimized Assembly implementation running on a 32-bit Cortex-M0+ processor clocked at 48 MHz that computes a point multiplication on a less secure Koblitz curve K-233 without strong side-channel countermeasures. It computes a point multiplication in 59.18 ms (177.54 ms at 16 MHz) and consumes 34.16 μJ of energy.

Table 3.1: Comparison to other lightweight coprocessors for ECC. The top part consists of relevant implementations from the literature. We also provide estimates for other parameter sets in order to ease comparisons to existing works.

Work	Curve	Conv.	RAM	Tech. (nm)	Freq. (MHz)	Area (GE)	Latency (cycles)	Latency (ms)	Power (μW[1])	Energy (μJ[1])
[13], 2006	B-163	n/a	no	130	0.500	9,926	95,159	190.32	<60	<5.7
[18], 2008	B-163	n/a	yes	220	0.847	12,876	–	95	93	7.48
[55], 2008	B-163	n/a	yes	180	0.106	13,250	296,299	2,792	80.85	23.9
[73], 2006	B-163	n/a	yes	350	13.560	16,207	376,864	27.90	n/a	n/a
[74], 2008	B-163	n/a	yes	130	1.130	12,506	275,816	244.08	32.42	8.94
[145], 2011	B-163	n/a	yes	130	0.100	8,958	286,000	2,860	32.34	9.25
[144], 2013	B-163	n/a	no	130	1.000	4,114	467,370	467.37	66.1	30.9
[100], 2014	P-160	n/a	yes	130	1.000	12,448[2]	139,930	139.93	42.42	5.93
[11], 2014	K-163	no	yes[3]	65	13.560	11,571	106,700	7.87	5.7	0.6
Our, est.	B-163	yes	no	130, Faraday	16.000	≈3,773	≈485,000	≈30.31	≈6.11	2.96
Our, est.	K-163	yes	no	130, Faraday	16.000	≈4,323	≈420,900	≈26.30	≈6.11	2.57
Our, est.	B-283	yes	no	130, Faraday	16.000	≈3,773	≈1,934,000	≈120.89	≈6.11	11.8
Our, est.	K-283	yes	yes[4]	130, Faraday	16.000	10,204	1,566,000	97.89	>6.11	>9.6
Our	**K-283**	**yes**	**no**	**130**	**16.000**	**4,323**	**1,566,000**	**97.89**	**6.11**	9.6

[1] Normalized to 1 MHz.
[2] Contains everything required for ECDSA including a Keccak module.
[3] All variables are stored in registers inside the processor.
[4] The 256 × 16-bit RAM is estimated to have an area of 5794 GE because the size of a single-port 256 × 8-bit RAM has an area of 2897 GE [144].
Note: For a similar technology, there can be small variations in area, power, and energy depending on the library.

3.6 Summary

In this chapter we showed that implementing point multiplication on a high security 283-bit Koblitz curve is feasible with extremely low resources making it possible for various lightweight applications. We also showed that Koblitz curves can be used in such applications even when the cryptosystem requires scalar conversions. Beside these contributions, we improved the scalar conversion by applying several optimizations and countermeasures against side-channel attacks. Finally, we designed a very lightweight architecture in only 4.3 kGE that can be used as a coprocessor for commercial 16-bit microcontrollers. Hence, we showed that Koblitz curves are feasible also for lightweight ECC even with on-the-fly scalar conversions and strong countermeasures against side-channel attacks.

Chapter 4

Discrete Gaussian sampling

CONTENT SOURCES:

Sujoy Sinha Roy, Frederik Vercauteren, Ingrid Verbauwhede High precision discrete Gaussian sampling on FPGAs In *International Conference on Selected Areas in Cryptography SAC* (2013).

Contribution: Main author.

Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, Ingrid Verbauwhede Compact and Side Channel Secure Discrete Gaussian Sampling In *IACR Cryptology ePrint Archive eprint/2014/591* (2014).

Contribution: Main author.

4.1 Introduction

In this chapter we propose an efficient hardware implementation of a discrete Gaussian sampler for ring-LWE encryption schemes. The proposed sampler architecture is based on the Knuth-Yao sampling Alg. [68]. It has high precision and large tail-bound to keep the statistical distance below 2^{-90} to the true Gaussian distribution for the secure parameter sets [52] that are used in the public key encryption schemes [111, 78].

The remaining part of the chapter is organized as follows: In Sect. 4.2 we describe the Knuth-Yao sampling algorithm in detail. In the next section we analyze the Knuth-Yao algorithm and design an efficient algorithm that

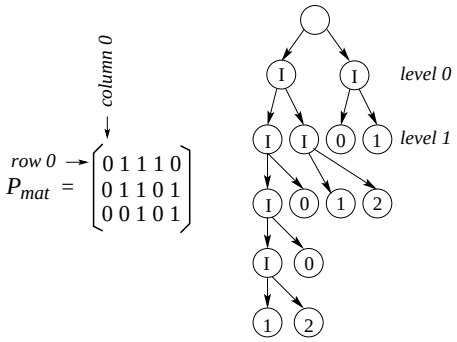

Figure 4.1: Probability matrix and corresponding DDG-tree

consumes very little amount of resources. The hardware architecture of the discrete Gaussian sampler is presented in Sect. 4.4. In Sect. 4.5 we describe side channel vulnerability of the sampler architecture along with countermeasures. Detailed experimental results are presented in Sect. 4.6. The final section has the conclusion.

4.2 The Knuth-Yao algorithm

The Knuth-Yao algorithm uses a random walk model to perform sampling using the probabilities of the sample space elements. The method is applicable for any non-uniform distribution. Let p_j be the probability of the jth sample in the sample space. The binary expansions of the probabilities of the samples are written in the form of a matrix which we call the *probability matrix* P_{mat}. The jth row of the probability matrix corresponds to the binary expansion of p_j. An example of the probability matrix for a sample space containing three sample points $\{0, 1, 2\}$ with probabilities $p_0 = 0.01110$, $p_1 = 0.01101$ and $p_2 = 0.00101$ is shown in Fig. 4.1.

A rooted binary tree known as a discrete distribution generating (DDG) tree is constructed from the probability matrix. Each level of the DDG tree can have two types of nodes: intermediate nodes (I) and terminal nodes. The number of terminal nodes in the ith level of the DDG tree is equal to the Hamming weight of ith column in the probability matrix. Here we provide an example of the DDG tree construction for the given probability distribution in Fig. 4.1. The root of the DDG tree has two children which form the 0th level. Both the nodes in this level are marked with I since the 0th column in P_{mat} does not

contain any non-zero. These two intermediate nodes have four children in the 1st level. To determine the type of the nodes, the 1st column of P_{mat} is scanned from the bottom. In this column only the row numbers '1' and '0' are non-zero; hence the right-most two nodes in the 1st level of the tree are marked with '1' and '0' respectively. The remaining two nodes in this level are thus marked as intermediate nodes. Similarly the next levels are also constructed. The DDG tree corresponding to P_{mat} is given in Fig. 4.1. At any level of the DDG tree, the terminal nodes (if present) are always on the right hand side.

The sampling operation is a random walk which starts from the root; visits a left-child or a right-child of an intermediate node depending on the random input bit. The sampling process completes when the random walk hits a terminal node and the output of the sampling operation is the value of the terminal node. By construction, the Knuth-Yao random walk samples accurately from the distribution defined by the probability matrix.

The space requirement for the DDG tree can be reduced by constructing it at run time during the sampling process. As shown in Fig. 4.1, the ith level of the DDG tree is completely determined by the $(i-1)$th level and the ith column of the probability matrix. Hence it is sufficient to store only one level of the DDG tree during the sampling operation and construct the next level on the fly (if required) using the probability matrix [68].

In fact, in the next section we introduce a novel method to traverse the DDG tree that only requires the current node and the ith column of the probability matrix to derive the next node in the tree traversal.

4.3 DDG tree on the fly

In this section we propose an efficient hardware-implementation of the Knuth-Yao based discrete Gaussian sampler which samples with high precision and large tail-bound. We describe how the DDG tree can be traversed efficiently in hardware and then propose an efficient way to store the probability matrix such that it can be scanned efficiently and also requires near-optimal space. Before we describe the implementation of the sampler, we first recall the parameter set for the discrete Gaussian sampler from the LWE implementation in [52].

4.3.1 Parameter sets for the discrete Gaussian sampler

Table 4.1 shows the tail bound $|z_t|$ and precision ϵ required to obtain a statistical distance of less than 2^{-90} for the Gaussian distribution parameters in Table 1

of [52]. The dimension of the lattice is m. The standard deviation σ in Table 4.1 is derived from the parameter s using the equation $s = \sigma\sqrt{2\pi}$. The tail bound $|z_t|$ is calculated from Eq. 2.7 for the right-hand upper bound 2^{-100}. For a maximum statistical distance of 2^{-90} and a tail bound $|z_t|$, the required precision ϵ is calculated using Eq. 2.8.

Table 4.1: Parameter sets to achieve statistical distance less than 2^{-90}

m	s	σ	Tail cut $\lvert z_t \rvert$	Precision ϵ
256	8.35	3.33	84	106
320	8.00	3.192	86	106
512	8.01	3.195	101	107
1024	8.01	3.195	130	109

However in practice the tail bounds are quite loose for the precision values in Table 4.1. The probabilities are zero (upto the mentioned precision) for the sample points greater than 39 for all three distributions. Given a probability distribution, the Knuth-Yao random walk always hits a sample point when the sum of the probabilities is one [68]. However if the sum is less than one, then the random walk may not hit a terminal node in the corresponding DDG tree. Due to finite range and precision in Table 4.1, the sum of the discrete Gaussian probability expansions (say P_{sum}) is less than one. We take an difference $(1 - P_{sum})$ as another sample point which indicates *out of range* event. If the Knuth-Yao random walk hits this sample point, the sample value is discarded. This *out of range* event has probability less than 2^{-100} for all three distribution sets.

4.3.2 Construction of the DDG tree during sampling

During the Knuth-Yao random walk, the DDG tree is constructed at run time. The implementation of DDG tree as a binary tree data structure is an easy problem [28] in software, but challenging on hardware platforms. As described in Sect. 4.2, the implementation of the DDG tree requires only one level of the DDG tree to be stored. However the ith level of a DDG tree may contain as many as 2^i nodes. On software platforms, dynamic memory allocation can be used at run time to allocate sufficient memory required to store a level of the DDG tree. But in hardware, we need to design the sampler architecture for the worst case storage requirement which makes the implementation costly.

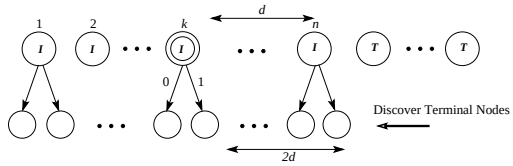

Figure 4.2: DDG Tree Construction

We propose a hardware implementation friendly traversal based on specific properties of the DDG tree. We observe that in a DDG tree, all the intermediate nodes are on the left hand side; while all the terminal nodes are on the right hand side. This observation is used to derive a simple algorithm which identifies the nodes in the DDG tree traversal path instead of constructing each level during the random walk. Fig. 4.2 describes the $(i-1)$th level of the DDG tree. The intermediate nodes are I, while the terminal nodes are T. The node visited at this level during the sampling process is highlighted by the double circle in the figure. Assume that the visited node is not a terminal node. This assumption is obvious because if the visited node is a terminal node, then we do not need to construct the ith level of the DDG tree. At this level, let there be n intermediate nodes and the visited node is the kth node from the left. Let $d = n - k$ denote the distance of the right most intermediate node from the visited node.

In the next step, the sampling algorithm reads a random bit and visits a child node on the ith level of the DDG tree. If the visited node is a left child, then it has $2d + 1$ nodes to its right side. Otherwise, it will have $2d$ nodes to its right side (as shown in the figure). To determine whether the visited node is a terminal node or an intermediate node, the ith column of the probability matrix is scanned. The scanning process detects the terminal nodes from the right side of the ith level and the number of terminal nodes is equal to the Hamming weight h of the ith column of the probability matrix. The left child is a terminal node if $h > (2d + 1)$ and the right child is a terminal node if $h > 2d$. If the visited node is a terminal node, we output the corresponding row number in the probability matrix as the result of sampling process. When the visited node in the ith level is internal, its visited-child in the $(i + 1)$th level is checked in a similar way.

From the analysis of DDG tree construction, we see the following points :

1. The sampling process is independent of the internal nodes that are to the left of the visited node.

Algorithm 7: *Knuth-Yao Sampling*

```
Input: Probability matrix P
Output: Sample value S
1  begin
2      d ← 0; /* Distance between the visited and the rightmost internal node */
3      Hit ← 0; /* This is 1 when the sampling process hits a terminal node */
4      col ← 0;     /* Column number of the probability matrix */
5      while Hit = 0 do
6          r ← RandomBit() ;
7          d ← 2d + r̄ ;
8          for row = MAXROW down to 0 do
9              d ← d − P[row][col] ;
10             if d = −1 then
11                 S ← row ;
12                 Hit ← 1 ;
13                 ExitForLoop() ;
14             end
15         end
16         col ← col + 1 ;
17     end
18 end
```

2. The terminal nodes on the $(i-1)$th level have no influence on the construction of the ith level of the DDG tree.

3. The distance d between the right most internal node and the visited node on the $(i-1)$th level of the DDG tree is sufficient (along with the Hamming weight of the ith column of the probability matrix) to determine whether the visited node on the ith level is an internal node or a terminal node.

During the Knuth-Yao sampling we do not store an entire level of the DDG tree. Instead, the difference d between the visited node and the right-most intermediate node is used to construct the visited node at the next level. The steps of the Knuth-Yao sampling operation are described in Alg. 7. In Line 6, a random bit r is used to jump to the next level of the DDG tree. On this new level, the distance between the visited node and the rightmost node is initialized to either $2d$ or $2d+1$ depending on the random bit r. In Line 8, the for-loop scans a column of the probability matrix to detect the terminal nodes. Whenever the algorithm finds a 1 in the column, it detects a terminal node. Hence, the relative distance between the visited node and the right most internal node is decreased by one (Line 9). When d is reduced to -1, the sampling algorithm hits a terminal node. Hence, in this case the sampling algorithm stops and returns the corresponding row number as the output. In the other case, when d is positive after completing the scanning of an entire column of the probability matrix, the sampling algorithm jumps to the next level of the DDG tree.

4.3.3 Storing the probability matrix efficiently

The Knuth-Yao algorithm reads the probability matrix of the discrete Gaussian distribution during formation of the DDG tree. A probability matrix having r rows and c columns requires rc bits of storage. This storage could be significant when both r (depends on the tail-bound) and c (depends on the precision) are large. As an example, Fig. 4.3 shows a portion of the probability matrix for the probabilities of $0 \leq |z| \leq 17$ with 30-bits precision according to the discrete Gaussian distribution with parameter $s = 8.01$. In [39] the authors observed that the leading zeros in the probability matrix can be compressed. The authors partitioned the probability matrix in different blocks having equal (or near-equal) number of leading zeros. Now for any row of the probability matrix, the conditional probability with respect to the block it belongs to is calculated and stored. In this case the conditional probability expansions do not contain a long sequence of leading zeros. The precision of the conditional probabilities is less than the precision of the absolute probabilities by roughly the number of leading zeros present in the absolute probability expansions. The sampling of [39] then applies two rounds of the Knuth-Yao algorithm: first to select a block and then to select a sample value according to the conditional probability expansions within the block.

However the authors of [39] do not give any actual implementation details. In hardware, ROM is ideal for storing a large amount of fixed data. To minimize computation time, data fetching from ROM should be minimized as much as possible. The pattern in which the probability expansions are stored in ROM determines the number of ROM accesses (thus performance) during the sampling process. During the sampling process the probability matrix is scanned column by column. Hence to ease the scanning operation, the probability expansions should be stored in a column-wise manner in ROM.

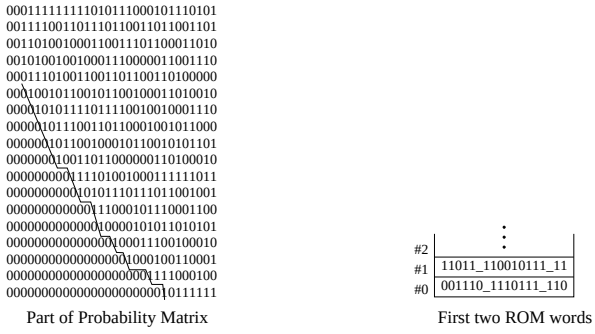

Figure 4.3: Storing Probability Matrix

In Fig. 4.3, the probability matrix for a discrete Gaussian distribution contains large chunks of zeros near the bottom of the columns. Since we store the probability matrix in a column-wise manner in ROM, we perform compression of zeros present in the columns. The *column length* is the length of the top portion after which the chunk of bottom zeros start. We target to optimize the storage requirement by storing only the upper portions of the columns in ROM. Since the columns have different lengths, we also store the lengths of the columns. The number of bits required to represent the length of a column can be reduced by storing only the difference in column length with respect to the previous column. In this case, the number of bits required to represent the differential column length is the number of bits in the maximum deviation and a sign bit. For the discrete Gaussian distribution matrix shown in Fig. 4.3, the maximum deviation is three and hence three bits are required to represent the differential column lengths. Hence the total number of bits required to store the differential column lengths of the matrix (Fig. 4.3) is 86 (ignoring the first two columns).

For the discrete Gaussian distribution matrix, we observe that the difference between two consecutive column lengths is one for most of the columns. This observation is used to store the distribution matrix more efficiently in ROM.

Algorithm 8: *Knuth-Yao Sampling in Hardware Platform*

Input: Probability matrix P
Output: Sample value S

```
1  begin
2      d ← 0;      /* Distance between the visited and the rightmost internal node */
3      Hit ← 0;      /* This is 1 when the sampling process hits a terminal node */
4      ColLen ← INITIAL;      /* Column length is set to the length of first column */
5      address ← 0;      /* This variable is the address of a ROM word */
6      i ← 0;      /* This variable points the bits in a ROM word */
7      while Hit = 0 do
8          r ← RandomBit() ;
9          d ← 2d + r̄ ;
10         ColLen ← ColLen + ROM[address][i] ;
11         for row = ColLen − 1 down to 0 do
12             i ← i + 1 ;
13             if i = w then
14                 address ← address + 1 ;
15                 i ← 0 ;
16             end
17             d ← d − ROM[row][i] ;
18             if d = −1 then
19                 S ← row ;
20                 Hit ← 1 ;
21                 ExitForLoop() ;
22             end
23         end
24     end
25     return (S)
26 end
```

We consider only non-negative differences between consecutive column lengths; the length of a column either increases or remains the same with respect to its left column. When there is a decrement in the column length, the extra zeros are also considered to be part of the column to keep the column length the same as its left neighbor. In Fig. 4.3 the dotted line is used to indicate the lengths of the columns. It can be seen that the maximum increment in the column length happens to be one between any two consecutive columns (except the initial few columns). In this representation only one bit per column is needed to indicate the difference with respect to the left neighboring column: 0 for no-increment and 1 for an increment by one. With such a representation, 28 bits are required to represent the increment of the column lengths for the matrix in Fig. 4.3. Additionally, 8 redundant zeros are stored at the bottom of the columns due to the decrease in column length in a few columns. Thus, a total of 36 bits are stored in addition to the pruned probability matrix. There is one more advantage of storing the probability matrix in this way in that we can use a simple binary counter to represent the length of the columns. The binary counter increments by one or remains the same depending on the column-length increment bit.

In ROM, we only store the portion of a column above the partition-line in Fig. 4.3 along with the column length difference bit. The column-length difference bit is kept at the beginning and then the column is kept in reverse order (bottom-to-top). As the Knuth-Yao algorithm scans a column from bottom to top, the column is stored in reverse order. Fig. 4.3 shows how the columns are stored in the first two ROM words (word size 16 bits). During the sampling process, a variable is used to keep track of the column-lengths. This variable is initialized to the length of the first non-zero column. For the probability matrix in Fig. 4.3, the initialization value is 5 instead of 4 as the length of the next column is 6. Whilst scanning a new column, this variable is either incremented (starting bit 1) or kept the same (starting bit 0). Alg. 8 summarizes the steps when a ROM of word size w is used as a storage for the probability matrix.

4.3.4 Fast sampling using a lookup table

A Gaussian distribution is concentrated around its center. In the case of a discrete Gaussian distribution with standard deviation σ, the probability of sampling a value larger than $t \cdot \sigma$ is less than $2\exp(-t^2/2)$ [84]. In fact this upper bound is not very tight. We use this property of a discrete Gaussian distribution to design a fast sampler architecture satisfying the speed constraints of many real-time applications. As seen from the previous section, the Knuth-Yao random walk uses random bits to move from one level of the DDG tree to the next level. Hence the average case computation time required per sampling

operation is determined by the number of random bits required in the average case.

The lower bound on the number of random bits required per sampling operation in the average case is given by the entropy of the probability distribution [33]. The entropy of a continuous normal distribution with a standard deviation σ is $\frac{1}{2}\log(2\pi e\sigma^2)$. For a discrete Gaussian distribution, the entropy is approximately close to entropy of the normal distribution with the same standard deviation. A more accurate entropy can be computed from the probability values as per the following equation.

$$H = -\sum_{-\infty}^{\infty} p_i \log p_i \ . \tag{4.1}$$

The Knuth-Yao sampling algorithm was developed to consume the minimum number of random bits on average [68]. It was shown that the sampling algorithm requires at most $H + 2$ random bits per sampling operation in the average case.

For a Gaussian distribution, the entropy H increases with the standard deviation σ, and thus the number of random bits required in the average case also increases with σ. For applications such as the ring-LWE based public key encryption scheme and homomorphic encryption, small σ is used. Hence for such applications the number of random bits required in the average case are small. Based on this observation we can avoid the costly bit-scanning operation using a small precomputed table that directly maps the initial random bits into a sample value (with large probability) or into an intermediate node in the DDG tree (with small probability). During a sampling operation, first a table lookup operation is performed using the initial random bits. If the table lookup operation returns a sample value, then the sampling algorithm terminates. For the other case, bit scanning operation is initiated from the intermediate node. For example, when $\sigma = 3.33$, if we use a precomputed table that maps the first eight random bits, then the probability of getting a sample value after the table lookup is 0.973. Hence using the lookup table we can avoid the costly bit-scanning operation with probability 0.973. However extra storage space is required for this lookup table. When the probability distribution is fixed, the lookup table can be implemented as a ROM which is cheap in terms of area in hardware platforms. In the next section we propose a cost effective implementation of a fast Knuth-Yao sampler architecture.

4.4 The sampler architecture

The sampler architecture is composed of 1) a bit-scanning unit, 2) counters for column length and row number, and 3) a subtraction-based down counter for the Knuth-Yao distance in the DDG tree. In addition, for the fast sampler architecture, a lookup table is also used. A control unit is used to generate control signals for the different blocks and to maintain synchronization between the blocks. We now describe the different components of the sampler architecture.

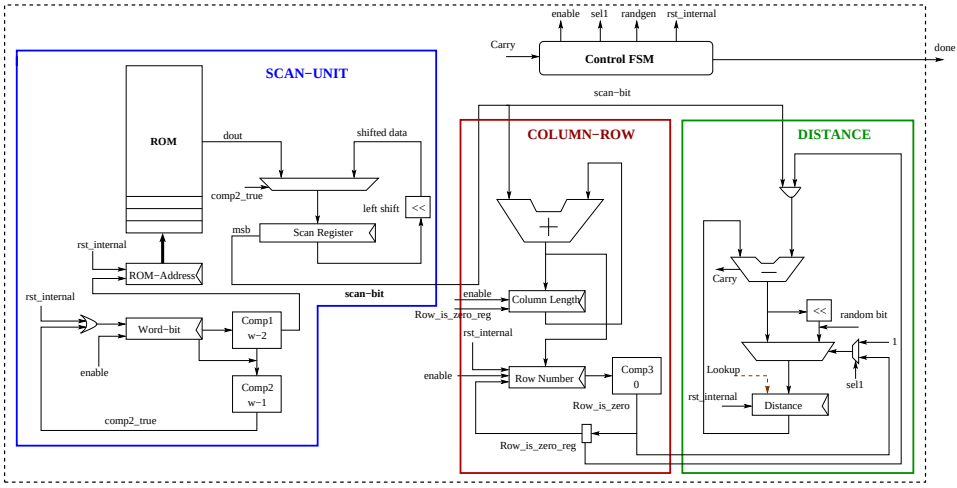

Figure 4.4: Hardware Architecture for Knuth-Yao Sampler

4.4.1 The bit-scanning unit

The bit-scanning unit is composed of a ROM, a scan register, one ROM-address counter, one counter to record the number of bits scanned from a ROM-word and a comparator. The ROM contains the probabilities and is addressed by the ROM-address counter. During a bit-scanning operation, a ROM-word (size w bits) is first fetched and then stored in the scan register. The scan-register is a shift-register and its msb is read as the probability-bit. To count the number of bits scanned from a ROM-word, a counter *word-bit* is used. When the *word-bit* counter reaches $w-2$ from zero, the output from the comparator *Comp1* enables the *ROM-address* counter. In the next cycle the *ROM-address* counter addresses the next ROM-word. Also in this cycle the *word-bit* counter reaches $w-1$ and

the output from *Comp2* enables reloading of the bit-scan register with the new ROM-word. In the next cycle, the *word-bit* counter is reset to zero and the bit-scan register contains the word addressed by the *ROM-word* counter. In this way data loading and shifting in the bit-scan register takes place without any loss of cycles. Thus the frequency of the data loading operation (which depends on the widths of the ROM) does influence the cycle requirement of the sampler architecture. This interesting feature of the bit-scan unit will be utilized in the next part of this section to achieve optimal area requirement by adjusting the width of the ROM and the bit-scan register. The bit-scanning unit is the largest sub-block in the sampler architecture in terms of area. Hence this unit should be designed carefully to achieve minimum area requirement. In FPGAs a ROM can be implemented as a distributed ROM or as a block RAM. When the amount of data is small, a distributed ROM is the ideal choice. The way a ROM is implemented (its width w and depth h) affects the area requirement of the sampler. Let us assume that the total number of probability bits to be stored in the ROM is D and the size of the FPGA LUTs is t. Then the total number of LUTs required by the ROM is around $\lceil \frac{D}{w \cdot 2^t} \rceil \cdot w$ along with a small amount of addressing overhead. The scan-register is a shift register of width w and consumes around w LUTs and $w_f = w$ FFs. Hence the total area (LUTs and FFs) required by the ROM and the scan-register can be approximated by the following equation.

$$\#Area \quad = \quad \lceil \frac{D}{w \cdot 2^t} \rceil \cdot w + (w + w_f) = \lceil \frac{h}{2^t} \rceil \cdot w + (w + w_f) \ .$$

For optimal storage, h should be a multiple of 2^t. Choosing a larger value of h will reduce the width of the ROM and hence the width of the scan-register. However with the increase in h, the addressing overhead of the ROM will also increase. In Table 4.2 we compare area of the bit-scan unit for $\sigma = 3.33$ with various widths of the ROM and the scan register using Xilinx Virtex V xcvlx30 FPGA. The optimal implementation is achieved when the width of the ROM is set to six bits. Though the slice count of the bit-scan unit remains the same in both the second and third column of the table due to various optimizations performed by the Xilinx ISE tool, the actual effect on the overall sampler architecture will be evident in Sect. 4.6.

Table 4.2: Area of the bit-scan unit for different widths and depths

width	*height*	*LUTs*	*FFs*	*Slices*
24	128	70	35	22
12	256	72	23	18
6	512	67	17	18

4.4.2 Row-number and column-length counters

As described in the previous section, we use a one-step differential encoding for the column lengths in the probability matrix. The *column-length* counter in Fig. 4.4 is an up-counter and is used to represent the lengths of the columns. During a random-walk, this counter increments depending on the column-length bit which appears in the starting of a column. If the column-length bit is zero, then the *column-length* counter remains in its previous value; otherwise it increments by one. At the starting of a column-scanning operation, the *Row-number* counter is first initialized to the value of column-length. During the scanning operation this counter decrements by one in each cycle. A column is completely read when the *Row Number* counter reaches zero.

4.4.3 The distance counter

A subtraction-based counter *distance* is used to keep the distance d between the visited node and the right-most intermediate node in the DDG tree. The register *distance* is first initialized to zero. During each column jump, the *row_zero_reg* is set and thus the subtrahend becomes zero. In this step, the *distance* register is updated with the value $2d$ or $2d + 1$ depending on the input random bit. As described in the previous section, a terminal node is visited by the random walk when the distance becomes negative for the first time. This event is detected by the control FSM using the carry generated from the subtraction operation.

After completion of a random walk, the value present in *Row Number* is the magnitude of the sample output. One random bit is used as a sign of the value of the sample output.

4.4.4 The lookup table for fast sampling

The output from the Knuth-Yao sampling algorithm is determined by the probability distribution and by the input sequence of random bits. For a given fixed probability distribution, we can precompute a table that maps all possible random strings of bit-width s into a sample point or into an intermediate distance in the DDG tree. The precomputed table consists of 2^s entries for each of the 2^s possible random numbers.

On FPGAs, this precomputed table is implemented as a distributed ROM using LUTs. The ROM contains 2^s words and is addressed by random numbers of s bit width. The success probability of a table lookup operation can be increased by increasing the size of the lookup table. For example when $\sigma = 3.33$, the

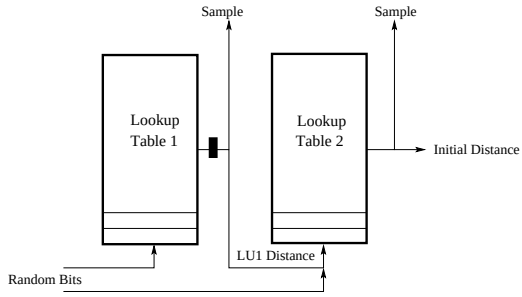

Figure 4.5: Hardware Architecture for two stage Lookup

probability of success is 0.973 when the lookup table maps the eight random bits; whereas the success probability increases to 0.999 when the lookup table maps 13 random bits. However with a larger mapping, the size of precomputed table increases exponentially from 2^8 to 2^{13}. Additionally each lookup operation requires 13 random bits. A more efficient approach is to perform lookup operations in steps. For example, we use a first lookup table that maps the first eight random bits into a sample point or an intermediate distance (three bit wide for $\sigma = 3.33$). In case of a lookup failure, the next step of the random walk from the obtained intermediate distance will be determined by the next sequence of random bits. Hence, we can extend the lookup operation to speedup the sampling operation. For example, the three-bit wide distance can be combined with another five random bits to address a (the *second*) lookup table. Using this two small lookup tables, we achieve a success probability of 0.999 for $\sigma = 3.33$. An architecture for a two stage lookup table is shown in Fig. 4.5.

4.5 Timing and simple power analysis

The Knuth-Yao sampler presented in this chapter is not a constant-time architecture as it uses a bit scanning operation in which the sample generated is related to the number of probability-bits scanned during a sampling operation. Hence, the number of cycles for a sampling operation provides some information about the sample. In 2016, Bruinderink et al. [53] mounted an attack on the BLISS [38] signature scheme by exploiting the timing information leakage from the non constant-time discrete Gaussian sampler. In particular, the attack targets a software implementation of the BLISS scheme and exploits cache weakness of the Gaussian sampler to run the LLL lattice reduction [75] algorithm. The attack requires as little as 450 signatures to recover the secrete key. The attack technique is described in more details in [53].

We recall that in the ring-LWE encryption scheme in Sect. 2.4.1, the Gaussian sampler is used during key generation and message encryption. The key generation operation is performed only to generate long-term keys and hence can be performed in a secure environment. However, this is not the case for the encryption operation where an encoded message $\bar{m}$ is masked as $c_2 = p \cdot e_1 + e_3 + \bar{m}$ using two Gaussian distributed error polynomials e_1 and e_3. It should be noted that in a public key encryption scheme, the plaintext is normally considered secret information. For example, it is a common practice to use a public-key cryptosystem to encrypt a symmetric key that is subsequently used for fast, bulk encryption (this construction is commonly named "hybrid cryptosystems"). Hence, from the perspective of side-channel analysis, any leak of information during the encryption operation about the plaintext (symmetric key) is considered as a valid security threat. We would like to mention that the timing attack proposed by Bruinderink et al. [53] on the BLISS signature scheme may not be applied directly on the ring-LWE public key encryption scheme as the attacker can have only one ciphertext per message. Whereas for the signature scheme, the attacker can use the LLL algorithm since she can have several signatures for the same secret key.

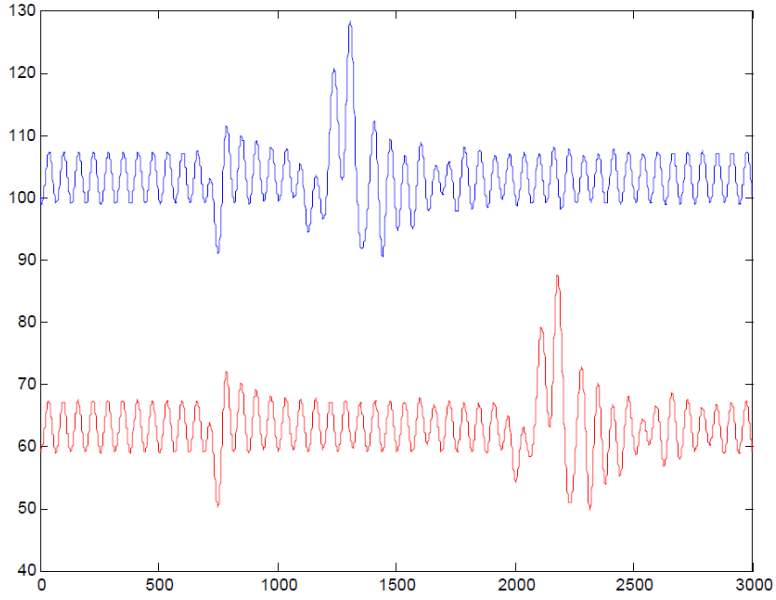

Figure 4.6: Two instantaneous power consumption measurements corresponding to two different sampling operations. Horizontal axis is time, vertical axis is electromagnetic field intensity. The different timing for the two different sampling operations is evident.

To verify to what extent the instantaneous power consumption provides information about the sampling operation, we performed an SPA attack on the unprotected design running on a Xilinx Spartan-III at 40 MHz. The instantaneous power consumption is measured with a Langer RF5-2 magnetic pick-up coil on top of the FPGA package (without decapsulation), amplified (+50 dB), low-pass filtered (cutoff frequency of 48 MHz). In Fig. 4.6 we show the instantaneous power consumption of two different sampling operations. The horizontal axis denotes time, and both sampling operations are triggered on the beginning of the sampling operation. One can distinguish enough SPA features (presumably due to register updates) to infer that the *blue* graph corresponds to a sampling that requires small number of cycles (7 cycles exactly) whereas the *red* graph represents a sampling operation that requires more cycles (21 cycles). With this SPA attack, the adversary can predict the *magnitudes* of the Gaussian distributed samples. Note that this information is partial as the value of a sample also includes a random sign bit. We remark that with a more sophisticated side channel attack, it might be possible to recover the sign bits by observing the modular arithmetic operations during the message encryption. If possible, then the message can be extracted easily from the ciphertext.

4.5.1 Strategies to mitigate the side-channel leakage

In this chapter we propose an efficient and cost effective scheme to protect the Gaussian sampler from simple timing and power analysis based attacks. Our proposal is based on the fact that the encryption scheme remains secure as long as the attacker has no information about the relative positions of the samples (i.e. the coefficients) in the noise polynomials. It should be noted that, as the Gaussian distribution used in the encryption scheme is a publicly known parameter, any one can guess the number of a particular sample point in an array of n samples. Similar arguments also apply for other cryptosystems where the key is a uniformly distributed random string of bits of some length (say l). For such a random key, one has the information that almost half of the bits in the key are one and the rest are zero. In other words, the Hamming weight is around $l/2$. Even if the exact value of the Hamming weight is revealed to the adversary (on average, say $l/2$), the key still mantains $\log_2\binom{l}{l/2}$ bits of entropy (≈ 124 bits for a 128 bit key). It is the random positions of the bits that make a key secure.

In the ring-LWE encryption scheme (Sect. 2.4.1), the Gaussian sampler is used to generate error polynomials. The sequential bit scanning operation reveals information about the samples and their positions in the error polynomials. Our strategy against simple timing and power analysis attack is described below:

1. Use of a lookup: The table lookup operation is constant-time and has a very large success probability. Hence with this lookup approach, we protect most of the samples from leaking any information about the value of the sample from which an attacker can perform simple power and timing analysis.

2. Use of a random permutation: The table lookup operation succeeds in most events, but fails with a small probability. For a failure, the sequential bit scanning operation leaks information about the samples. For example, when $\sigma = 3.33$ and the lookup table maps initial eight random bits, the bit scanning operation is required for seven samples out of 256 samples in the average case. To protect against SPA, we perform a random shuffle after generating an entire array of samples. The random shuffle operation swaps all bit-scan operation generated samples with other random samples in the array. This random shuffling operation removes any timing information which an attacker can exploit. In the next section we will describe an efficient implementation of the random shuffling operation.

4.5.2 Efficient implementation of the random shuffling

We use a modified version of the Fisher and Yates shuffle which is also known as the *Knuth shuffle* [67] to perform random shuffling of the bit-scan operation generated samples. The advantages of this shuffling algorithm are its simplicity, uniformness, inplace data handling and linear time complexity. In the original shuffling algorithm, all the indexes of the input array are processed one after another. However in our case we can restrict the shuffling operation to only those samples that were generated using the sequential bit scanning operation. This operation is implemented in the following way.

Algorithm 9: *Random swap of samples*

Input: Sample vector stored in RAM[ ] with timing information
Output: Sample vector stored in RAM[ ] without timing information

```
1  begin
2      while C2 > 0 do
3          L1 : random_index ← random() ;
4          if random_index ≥ (m − C2) then
5              goto L1 ;
6          end
7          swap RAM[m − C2] ↔ RAM[random_index] ;
8          C2 ← C2 − 1 ;
9      end
10 end
```

Assume that m samples are generated and then stored in a RAM with addresses in the range 0 to $(m-1)$. We use two counters C_1 and C_2 to represent

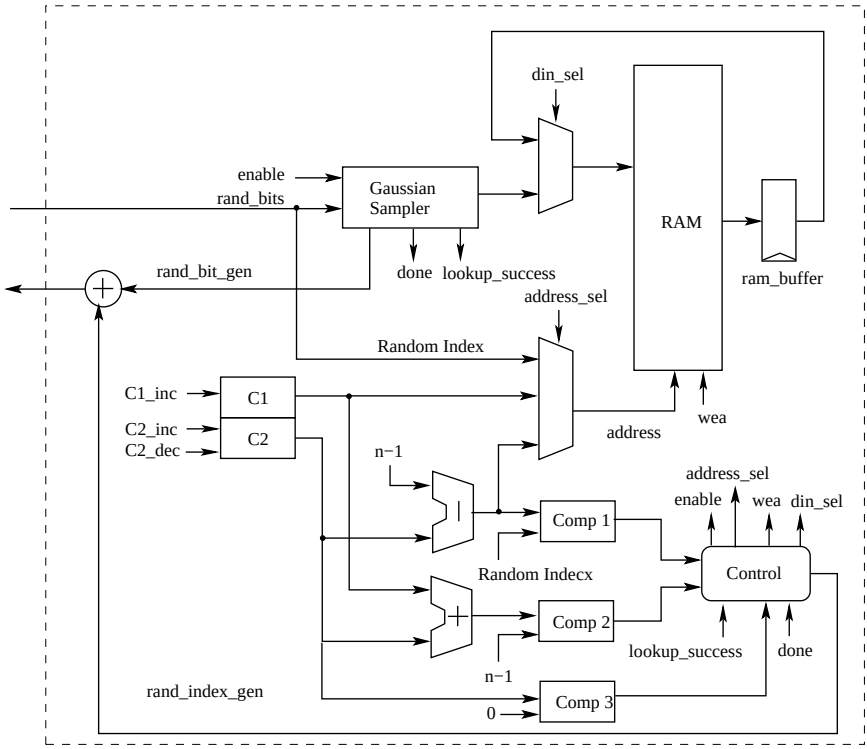

Figure 4.7: Sampler with shuffling

the number of samples generated through successful lookup and bit-scanning operations respectively. The total number of samples generated is given by $(C_1 + C_2)$. The samples generated using lookup operation are stored in the memory locations starting from 0 till $(C_1 - 1)$; whereas the bit-scan generated samples are stored in the memory locations starting from address $m - 1$ downto $m - C_2$. After generation of the m samples, the bit-scan operation generated samples are randomly swapped with the other samples using Alg. 9

A hardware architecture for the secure consecutive-sampling is shown in Fig. 4.7. In the architecture, C_1 is an up-counter and C_2 is an up-down-counter. When the *enable* signal is high, the Gaussian sampler generates samples in an iterative way. After generation of each sample, the signal *Gdone* goes high and the type of the sample is indicated by the signal *lookup_success*. In the case when the sample has been generated using a successful lookup operation, *lookup_success* becomes high. Depending on the value of the *lookup_success*, the control machine stores the sample in the memory address C_1 or $(m - C_2)$ and also

Table 4.3: Performance of the discrete Gaussian sampler on xc5vlx30

Sampler Architecture	ROM width/depth	LU depth	Area LUTs/FFs/Slices/BRAM	Delay ns	Cycles
Basic Sampler	24/128	-	101/81/38/-	2.9	17
Basic Sampler	12/256	-	105/60/32/-	2.5	17
Basic Sampler$\star$	6/512	-	102/48/30/-	2.6	17
Fast Sampler	6/512	8	118/48/35/-	3	$\approx$2.5
Bernoulli[105]	-/-	-	132/40/37/-	7.3	144
Polynomial Sampler–1	6/512	8	135/56/44/1	3.1	392
Polynomial Sampler–2	6/512	8	176/66/52/1	3.3	420

increments the corresponding counter. Completion of the m sampling operations is indicated by the output from *Comparator2*.

In the random-shuffling phase, a random address is generated and then compared with $(m - C_2)$. If the random-address is smaller than $(m - C_2)$ then it is used for the swap operation; otherwise another random-address is generated. Now the memory content of address $(m - C_2)$ is swapped with the memory content of random-address using the *ram_buffer* register. After this swap operation, the counter C_2 decrements by one. The last swap operation happens when C_2 is zero.

4.6 Experimental results

We have evaluated the Knuth-Yao discrete Gaussian sampler architecture for $\sigma = 3.33$ using the Xilinx Virtex V FPGA xcvlx30 with speed grade -3. The results shown in Table 4.3 are obtained from the Xilinx ISE12.2 tool after place and route analysis. In the table we show area and timing results of our architecture for various configurations and modes of operations and compare the results with other existing architectures. The results do not include the area of the random bit generator. Area requirements for the basic bit-scan operation based Knuth-Yao sampler for different ROM-widths and depths are shown in the first three rows of the table. The optimal area is achieved when the ROM-width is set to 6 bits. As the width of the ROM does not affect the cycle requirement of the sampler architecture, all different configurations have same clock cycle requirement. The average case cycle requirement of the sampler is determined by the number of bits scanned on average per sampling operation. A C program simulation shows that the number of memory-bits scanned on average is 13.5. Before starting the bit-scanning operation, the sampler performs two column jump operations for the first two all-zero columns of the probability matrix (for $\sigma = 3.33$). This initial operation requires two cycles. After this, the bit scan operation requires 14 cycles to scan 14 memory-bits and the final transition to

the completion state of the FSM requires one cycle. Thus, on average 17 cycles are spent per sampling operation. The compact Bernoulli sampler proposed in [105] consumes 37 slices and spends on average 144 cycles to generate a sample point.

The fast sampler architecture in the fourth column of Table 4.3 uses a lookup table that maps eight random bits. The sampler consumes additional five slices compared to the basic bit-scan based architecture. The probability that a table lookup operation returns a sample is 0.973. Due to this high success rate of the lookup operation, the average case cycle requirement of the fast sampler is slightly larger than 2 cycles with the consideration that one cycle is consumed for the transition of the state-machine to the completion state. In this cycle count, we assume that the initial eight random bits are available in parallel during the table lookup operation. If the random number generator is able to generate only one random bit per cycle, then additional eight cycles are required per sampling operation. However generating many (pseudo)random bits is not a problem using light-weight pseudo random number generators such as the trivium steam cipher which is used in [105]. The results in Table 4.3 show that by spending additional five slices, we can reduce the average case cycle requirement per sampling operation to almost two cycles from 17 cycles. As the sampler architecture is extremely small even with the lookup table, the acceleration provided by the fast sampling architecture will be useful in designing fast cryptosystems.

The Polynomial Sampler–1 of Table 4.3 generates a polynomial of $m = 256$ coefficients sampled from the discrete Gaussian distribution by using the fast sampler iteratively. The samples are stored in the RAM from address 0 to $m-1$. During the consecutive sampling operations, the state-machine jumps to the next sampling operation immediately after completing a sampling operation. In this consecutive mode of sampling operations, the 'transition to the end state' cycle is not spent for the individual sampling operations. As the probability of a successful lookup operation is 0.973, in the average case 249 out of the 256 samples are generated using successful lookup operations; whereas the seven samples are obtained through the sequential bit-scanning operation. In this consecutive mode of sampling, each lookup operation generated sample consumes one cycle. Hence in the average case 249 cycles are spent for generating the majority of the samples. The seven sampling operations that perform bit scanning starting from the ninth column of the probability matrix require on average a total of 143 cycles. Thus in total 392 cycles are spent on average to generate a Gaussian distributed polynomial.

The Polynomial Sampler–2 architecture includes the random shuffling operation on a Gaussian distributed polynomial of $m = 256$ coefficients. The architecture is thus secure against simple time and power analysis attacks. However this

security comes at the cost of an additional eight slices due to the requirement of additional counter and comparator circuits. The architecture first generates a polynomial in 392 cycles and then performs seven swap operations in 28 cycles in the average case. Thus in total the proposed side channel attack resistant sampler spends 420 cycles to generate a secure Gaussian distributed polynomial of 256 coefficients.

4.7 Summary

In this chapter we presented an optimized instance of the Knuth-Yao sampling architecture that consumes very small area. We showed that by properly tuning the width of the ROM and the scan register, and by a decentralizing the control logic, we can reduce the area of the sampler to only 30 slices without affecting the cycle count. Moreover, we proposed a fast sampling method using a very small-area precomputed table that reduces the cycle requirement by seven times in the average case. We showed that the basic sampler architecture can be attacked by exploiting its timing and power consumption related leakages. In the end we proposed a cost-effective counter measure that performs random shuffling of the samples.

Followup works. In joint works (coauthorship) with de Clercq et al. [31] and Liu et al. [81], we adapted our approach and implemented the software versions of the Knuth-Yao algorithm on 32-bit ARM and 8-Bit AVR processors respectively. We found that the Knuth-Yao algorithm performs equally well on the software platforms.

Bruinderink et al. [53] showed that the timing leakage from a non constant-time Gaussian sampler could be exploited to break signature schemes. The work mentions that the shuffling method (Sect. 4.5.1) increases the complexity of their attack.

Pessel [99] analyzed our shuffling based countermeasure in detail and proposed a profiled side channel attack that can recover the key by observing only 7,000 signatures. He proposed to use Gaussian convolution in conjunction with shuffling to increase side channel resistance.

In a joint work with Karmakar et al. [65] (coauthorship, under review), we have proposed a constant-time implementation of the Knuth-Yao sampling algorithm. Since the Knuth-Yao random walk is dictated by a set of input random bits, we could express the sample as a function of the input random bits. Hence we represent each bit of the output sample as a Boolean expression of the random input bits. During a sampling operation these Boolean expressions are evaluated

in constant-time and hence the computation time does not vary. To increase throughput, we use bit-slicing to generate multiple samples in batches.

Chapter 5

Ring-LWE public key encryption processor

CONTENT SOURCES:

Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede Compact ring-LWE cryptoprocessor In *International Workshop on Cryptographic Hardware and Embedded Systems CHES* (2014).

Contribution: Main author.

5.1 Introduction

In this chapter we analyze the LPR ring-LWE public key encryption scheme of Sect. 2.4.1 and design a compact hardware architecture of the encryption processor. From Fig. 2.5 of Sect. 2.4.1, we see that the LPR encryption scheme is composed of a discrete Gaussian sampler, a polynomial arithmetic (addition/multiplication) unit, a message encoder and a message decoder. In the last chapter we described how to design the discrete Gaussian sampler efficiently. In this chapter we first design a novel polynomial arithmetic unit and integrate it with the discrete Gaussian sampler to realize the ring-LWE public key encryption processor.

The polynomial arithmetic is computed in a ring $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f(x)\rangle$, where

one typically chooses $f(x) = x^n + 1$ with n a power of two, and q a prime with $q \equiv 1 \bmod 2n$. An implementation thus requires the basic operations in such a ring R_q, with multiplication taking up the bulk of the resources both in area and time. An efficient polynomial multiplier architecture therefore is a pre-requisite for the deployment of ring-LWE based cryptography in real world systems. In this chapter we investigate techniques to optimize the polynomial multiplication operation in R_q.

When we started this research, only a few hardware implementations [10, 52, 104, 106] of polynomial multipliers over R_q were known. All of these implementations have used the Number Theoretic Transform (NTT) to perform polynomial multiplication in R_q efficiently. It is well known that the Fast Fourier Transform (FFT) is asymptotically the fastest algorithm for computing polynomial multiplication [28]. NTT corresponds to a FTT where the roots of unity are taken from a finite ring instead of the complex numbers. Hence in an NTT all computations are performed on integers. The first published hardware implementation of ring-LWE encryption scheme [52] by Göttert et al. uses a fully parallel NTT structure for the polynomial multiplier resulting in a huge area consumption. For instance, even for medium security, their implementation does not fit on the largest FPGA of the Virtex 6 family. The later works [104, 106, 10] follow a sequential design methodology and use the FPGA resources in an efficient way.

In this chapter we analyze the NTT algorithm and propose several optimizations to reduce pre-computation overhead, memory requirement, and the number of memory access. We apply these optimization techniques to design a compact polynomial arithmetic core. We also perform several architectural optimizations to improve the operating frequency. Finally we connect the polynomial arithmetic core with the discrete Gaussian sampler and a memory bank to design a compact and efficient ring-LWE public key encryption processor.

The remainder of the chapter is organized as follows: In Sect. 5.2 we briefly describe the NTT algorithm and its application in computing polynomial multiplication. Sect. 5.3 contains our optimization techniques of the NTT and Sect. 5.4 presents the actual architecture of our optimized NTT algorithm. A pipelined architecture is given in Sect. 5.5. In Sect. 5.6, we propose an optimization of an existing ring-LWE encryption scheme and propose an efficient architecture for the complete ring-LWE encryption system. Sect. 5.7 reports on the experimental results of this implementation.

Target parameter sets

We have chosen to instantiate the cryptoprocessor for the parameter sets (n, q, s) (recall $s = \sqrt{2\pi}\sigma$), namely $P_1 = (256, 7681, 11.32)$ and $P_2 = (512, 12289, 12.18)$. Note that the choice of primes is not optimal for fast modular reduction. To estimate the security level offered by these two parameter sets we follow the security analysis in [80] and [76] which improves upon [78, 136]. Apart from the dimension n, the hardness of the ring-LWE problem mainly depends on the ratio q/σ, where clearly the problem becomes easier for larger ratios. Although neither parameter set was analyzed in [80], parameter set P_1 is similar to the set $(256, 4093, 8.35)$ from [80] which requires 2^{105} seconds to break, or still over 2^{128} elementary operations. For paramater set P_2 we expect it to offer a high security level consistent with AES-256 (following [52]).

We limit the Gaussian sampler in our implementation to 12σ to obtain a negligible statistical distance ($< 2^{-90}$) from the true discrete Gaussian distribution. Although one can normally sample the secret $r_2 \in R_q$ also from the distribution $\mathcal{X}_\sigma$, we restrict r_2 to have binary coefficients.

5.2 Polynomial multiplication

Recall from Chap. 2.5.2 that NTT leads to a fast multiplication algorithm in the ring $S_q = \mathbb{Z}_q[x]/(x^n - 1)$: indeed, given two polynomials $a, b \in S_q$ we can easily compute their (reduced) product $c = a \cdot b \in S_q$ by computing

$$c = NTT^{-1}_{\omega_n}\left(NTT_{\omega_n}(a) * NTT_{\omega_n}(b)\right), \tag{5.1}$$

where $*$ denotes point-wise multiplication.

The NTT computation is usually described as recursive, but in practice we use an in-place iterative version taken from [28] that is given in Alg. 10. For the inverse NTT, an additional scaling of the resulting coefficients by n^{-1} is performed. The factors ω used in line 8 are called the *twiddle factors*.

Multiplication in R_q:

Recall that we will use $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$ with $f = x^n + 1$ and $n = 2^k$. Since $f(x)|x^{2n} - 1$ we could use the $2n$-point NTT to compute the multiplication in R_q at the expense of three $2n$-point NTT computations and a reduction by trivially embedding the ring R_q into S_q, i.e. expanding the coefficient vector of

Algorithm 10: *Iterative NTT*

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \text{NTT}(a)$

```
1  begin
2      A ← BitReverse(a);
3      for m = 2 to n by m = 2m do
4          ω_m ← ω_n^{n/m} ;
5          ω ← 1 ;
6          for j = 0 to m/2 − 1 do
7              for k = 0 to n − 1 by m do
8                  t ← ω · A[k + j + m/2] ;
9                  u ← A[k + j] ;
10                 A[k + j] ← u + t ;
11                 A[k + j + m/2] ← u − t ;
12             end
13             ω ← ω · ω_m ;
14         end
15     end
16 end
```

a polynomial $a \in R_q$ by adding n extra zero coefficients. However, we can do much better by exploiting the special relation between the roots of $x^n + 1$ and $x^{2n} - 1$ using a technique known as the *negative wrapped convolution*.

Indeed, using the same evaluation-interpolation strategy used above for the ordinary NTT, we conclude that we can efficiently multiply two polynomials $a, b \in R_q$ if we can quickly evaluate them in the roots of f. These roots are simply ω_{2n}^{2j+1} for $j = 0, \ldots, n-1$ (since the even exponents give the roots of $x^n - 1$) and as such can be written as $\omega_{2n} \cdot \omega_n^j$. These evaluations can thus be computed efficiently using a *classical* n-point NTT (instead of a $2n$-point NTT) on the scaled polynomials $a'(x) = a(\omega_{2n} \cdot x)$ and $b'(x) = a(\omega_{2n} \cdot x)$. The point-wise multiplication gives the evaluations of $c(x) = a(x)b(x) \bmod f(x)$ in the roots of f, and the classical inverse n-point NTT thus results in the coefficients of the scaled polynomial $c'(x) = c(\omega_{2n} \cdot x)$. To recover the coefficients c_i of $c(x)$, we therefore simply have to compute $c_i = c'_i \cdot \omega_{2n}^{-i}$. Note that the scaling operation by n^{-1} can be combined with the multiplications of c'_i by ω_{2n}^{-i}.

5.3 Optimization of the NTT computation

In this section we optimize the NTT and compare with the recent hardware implementations of polynomial multipliers [10, 104, 106]. First, the fixed cost involved in computing the powers of ω_n is reduced, then the pre-computation overhead in the forward negative-wrapped convolution is optimized, and finally an efficient memory access scheme is proposed that reduces the number of

memory accesses during the NTT and also minimizes the number of block RAMs in the hardware architecture.

5.3.1 Optimizing the fixed computation cost

In line 13 of Alg. 10 the computation of the twiddle factor $\omega \leftarrow \omega \cdot \omega_m$ is performed in the j-loop. This computation can be considered as a fixed cost. However in [10, 104] the j-loop and the k-loop are interchanged, such that ω is updated in the innermost loop which is much more frequent than in Alg. 10. To avoid the computation of the twiddle factors, in [104] all the twiddle factors are kept in a pre-computed look-up table (ROM) and are accessed whenever required. As the twiddle factors are not computed on-the-fly, the order of the two innermost loops does not result in an additional cost. However in [10] a more compact polynomial multiplier architecture is designed without using any look-up table and the twiddle factors are simply computed on-the-fly during the NTT computation. Hence in [10], the interchanged loops cause substantial additional computational overhead. In this chapter our target is to design a very compact polynomial multiplier. Hence we do not use any look-up table for the twiddle factors and follow Alg. 10 to avoid the extra computation of [10].

5.3.2 Optimizing the forward NTT computation cost

Here we revisit the forward negative-wrapped convolution technique used in [10, 104, 106]. Recall that the negative-wrapped convolution corresponds to a classical n-point NTT on the scaled polynomials $a'(x) = a(\omega_{2n} \cdot x)$ and $b'(x) = (\omega_{2n} \cdot x)$. Instead of first pre-computing these scaled polynomials and then performing a classical NTT, it suffices to note that we can integrate the scaling and the NTT computation. Indeed, it suffices to change the initialization of the twiddle factors in line 5 of Alg. 10: instead of initializing ω to 1, we can simply set $\omega = \omega_{2m}$. The rest of the algorithm remains exactly the same, and no pre-computation is necessary. Note that this optimization only applies to the NTT itself and not to the inverse NTT.

5.3.3 Optimizing the memory access scheme

The NTT computation requires memory to store the input and intermediate coefficients. When the number of coefficients is large, RAM is most suitable for hardware implementation [10, 104, 106]. In the innermost loop (lines 8-to-11) of Alg. 10, two coefficients $A[k + j]$ and $A[k + j + m/2]$ are first read from

memory and then arithmetic operations (one multiplication, one addition and one subtraction) are performed. The new $A[k+j]$ and $A[k+j+m/2]$ are then written back to memory. During one iteration of the innermost loop, the arithmetic circuits are thus used only once, while the memory is read and written twice. This leads to idle cycles in the arithmetic circuits. The polynomial multiplier in [104] uses two parallel memory blocks to provide a continuous flow of coefficients to the arithmetic circuits. However this approach could result in under-utilization of the RAM blocks if the coefficient size is much smaller than the word size (for example in the ring-LWE cryptosystem [85]). In the literature there are many papers on efficient memory management schemes using segmentation and efficient address generation (see [86]) for the classical FFT algorithm. Another well known approach is the constant geometry FFT (or NTT) which always maintains a constant index difference between the processed coefficients [101]. However the constant geometry algorithm is not in-place and hence not suitable for resource constrained platforms. In [10] memory usage is improved by keeping two coefficients $A[k]$ and $B[k]$ of the two input polynomials A and B in the same memory location. We propose a memory access scheme which is designed to minimize the number of block RAM slices and to achieve maximum utilization of computational circuits present in the NTT architecture.

Since the two coefficients $A[k+j]$ and $A[k+j+m/2]$ are processed together in Alg. 10, we keep the two coefficients as a pair in one memory location.

Let us analyze two consecutive iterations of the m-loop (line 3 in Alg. 10) for $m = m_1$ and $m = m_2$ where $m_2 = 2m_1$. In the m_1-loop, for some j_1 and k_1 (maintaining the loop bounds in Alg. 10) the coefficients $(A[k_1 + j_1], A[k_1 + j_1 + m_1/2])$ are processed as a pair. Then k increments to $k_1 + m_1$ and the processed coefficient pair is $(A[k_1 + m_1 + j_1], A[k_1 + m_1 + j_1 + m_1/2])$. Now from Alg. 10 we see that the coefficient $A[k_1 + j_1]$ will again be processed in the m_2-loop with coefficient $A[k_1 + j_1 + m_2/2]$. Since $m_2 = 2m_1$, the coefficient $A[k_1 + j_1 + m_2/2]$ is the coefficient $A[k_1 + j_1 + m_1]$ which is updated in the m_1-loop for $k = k_1 + m_1$. Hence during the m_1-loop if we swap the updated coefficients for $k = k_1$ and $k = k_1 + m_1$ and store $(A[k_1 + j_1], A[k_1 + j_1 + m_1])$ and $(A[k_1 + j_1 + m_1/2], A[k_1 + j_1 + 3m_1/2])$ as the coefficient pairs in memory, then the coefficients in a pair have a difference of $m_2/2$ in their index and thus are ready for the m_2-loop. The operations during the two consecutive iterations $k = k_1$ and $k = k_1 + m_1$ during $m = m_1$ are shown in Alg. 11 in lines 8-15. During the operations u_1, t_1, u_2 and t_2 are used as temporary storage registers.

A complete description of the efficient memory access scheme is given in Alg. 11. In this algorithm for all values of $m < n$, two coefficient pairs are processed in the innermost loop and a swap of the updated coefficients is performed before writing back to memory. For $m = n$, no swap operation is required as this is the final iteration of the m-loop. The coefficient pairs generated by Alg. 11 can

Algorithm 11: *Iterative NTT : Memory Efficient Version*

Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and n-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \text{NTT}(a)$

```
1  begin
2      A ← BitReverse(a); /* Coefficients are stored in the memory as proper pairs */
3      for m = 2 to n/2 by m = 2m do
4          ω_m ← m-th primitiveroot(1) ;
5          ω ← squareroot(ω_m) or 1 /* Depending on forward or backward NTT */ ;
6          for j = 0 to m/2 − 1 do
7              for k = 0 to n/2 − 1 by m do
8                  (t_1, u_1) ← (A[k + j + m/2], A[k + j]) /* From MEMORY[k+j] */ ;
9                  (t_2, u_2) ← (A[k + m + j + m/2], A[k + m + j]) /* MEMORY[k+j+m/2] */ ;
10                 t_1 ← ω · t_1 ;
11                 t_2 ← ω · t_2 ;
12                 (A[k + j + m/2], A[k + j]) ← (u_1 − t_1, u_1 + t_1) ;
13                 (A[k + m + j + m/2], A[k + m + j]) ← (u_2 − t_2, u_2 + t_2) ;
14                 MEMORY[k + j] ← (A[k + j + m], A[k + j]) ;
15                 MEMORY[k + j + m/2] ← (A[k + j + 3m/2], A[k + j + m/2]) ;
16             end
17             ω ← ω · ω_n ;
18         end
19     end
20     m ← n ;
21     k ← 0 ;
22     ω ← squareroot(ω_m) or 1 /* Depending on forward or backward NTT */ ;
23     for j = 0 to m/2 − 1 do
24         (t_1, u_1) ← (A[j + m/2], A[j]) /* From MEMORY[j] */ ;
25         t_1 ← ω · t_1 ;
26         (A[j + m/2], A[j]) ← (u_1 − t_1, u_1 + t_1) ;
27         MEMORY[j] ← (A[j + m/2], A[j]) ;
28         ω ← ω · ω_m ;
29     end
30 end
```

be re-arranged easily for another (say inverse) NTT operation by performing address-wise bit-reverse-swap operation. Appendix A describes the memory access scheme using an example.

5.4 The NTT processor organization

In this section we present an architecture for performing the forward and backward NTT using the proposed optimization techniques. Our NTT processor (Fig. 5.1) consists of three main components: the arithmetic unit, the memory block and the control-address unit.

The memory block

is implemented as a simple dual port RAM. To accommodate two coefficients, the word size is $2\lceil \log q \rceil$ where q is the prime modulus. For the chosen parameter sets, coefficients are 13-bit or 14-bit wide. In FPGAs, a RAM can be implemented as a *distributed* or as a *block* RAM. When the amount of data is large, block RAM is the ideal choice.

The arithmetic unit (NTT-ALU)

is designed to support Alg. 11 along with other operations such as polynomial addition, point-wise multiplication and rearrangement of the coefficients. This NTT-ALU is interfaced with the memory block and the control-address unit. The central part of the NTT-ALU consists of a modular multiplier and addition/subtraction circuits.

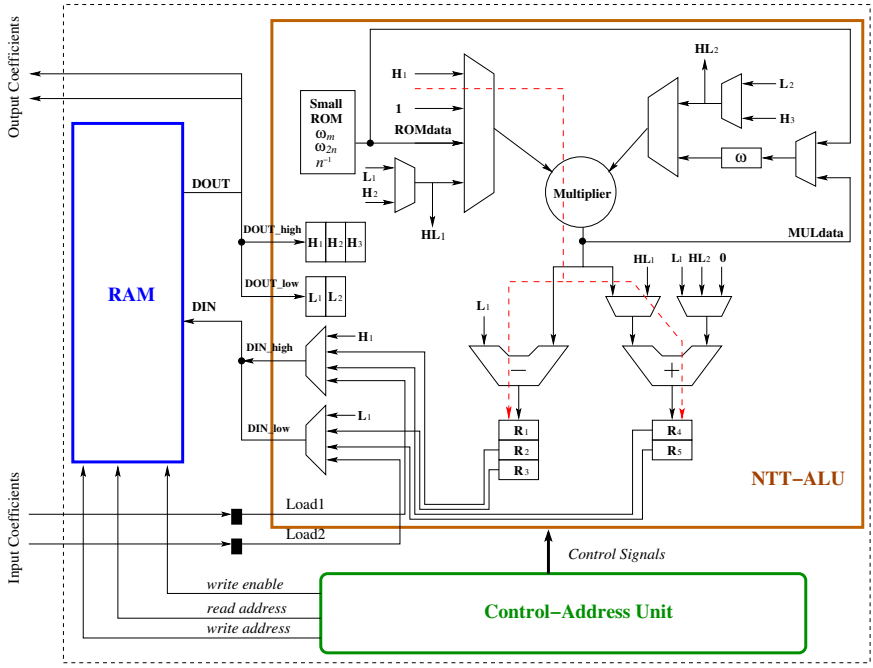

Figure 5.1: Hardware Architecture for NTT

Now we describe how the different components of the NTT-ALU are used during the butterfly steps (excluding the last loop for $m = n$).

1. First, the memory location $(k + j)$ is fetched and then the fetched data (t_1, u_1) is stored in the input register pair (H_1, L_1).
2. The same also happens for the memory location $(k + j + m/2)$ in the next cycle.
3. The multiplier computes $\omega \cdot H_1$ and the result is added to or subtracted from L_1 using the adder and subtracter circuits to compute $(u_1 + \omega t_1)$ and $(u_1 - \omega t_1)$ respectively.
4. In the next cycle the register pair (R_1, R_4) is updated with $(u_1 - \omega t_1, u_1 + \omega t_1)$.
5. Another clock transition shifts the contents of (R_1, R_4) to (R_2, R_5). In this cycle the pair (R_1, R_4) is updated with $(u_2 - \omega t_2, u_2 + \omega t_2)$ as the computation involving (u_2, t_2) from the location $(k + j + m/2)$ lags by one cycle.
6. Now the memory location $(k+j)$ is updated with the register pair (R_4, R_5) containing $(u_2 + \omega t_2, u_1 + \omega t_1)$.
7. Finally, in the next cycle the memory location $(k + j + m/2)$ is updated with $(u_2 - \omega t_2, u_1 - \omega t_1)$ using the register pair (R_2, R_3).

The execution of the *last* m-loop is similar to the intermediate loops, without any data swap between the output registers. The register pair (R_2, R_5) is used for updating the memory locations. In Fig. 5.1, the additional registers (H_2, H_3 and L_2) and multiplexers are used for supporting operations such as addition, point-wise multiplication and rearrangement of polynomials. The Small-ROM block contains the fixed values ω_m, ω_{2n}, their inverses and n^{-1}. This ROM has depth of order $\log(n)$.

The control-and-address Unit

consists of three counters for m, j and k in Alg. 11 and comparators to check the terminal conditions during the execution of any loop. The read address is computed from m, j and k and then delayed using registers to generate the write address. The control-and-address unit also generates the write enable signal for the RAM and the control signals for the NTT-ALU.

5.5 Pipelining the NTT processor

The maximum frequency of the NTT-ALU is determined by the critical path (red dashed line in Fig. 5.1): it passes through the modular multiplier and the adder (or subtracter) circuits . To increase the operating frequency of the processor, we implement efficient pipelines based on the following two observations.

Observation 1: During the execution of any m-loop in Alg. 11, the computations (multiplication, addition and subtraction) involving a coefficient pair have no data dependency on other coefficient pairs. Such a data-flow structure is suitable for pipeline processing as different computations can be pipelined without inserting bubbles in the datapath.

Assume that the modular multiplier has d_m pipeline stages and that the output is latched in a buffer. In the $(d_m + 1)$th cycle after the initialization of $\omega \cdot t_1$, the buffer is updated with the result $\omega \cdot t_1$. Now we need to compute $u_1 + \omega \cdot t_1$ and $u_1 - \omega \cdot t_1$ using the adder and subtracter circuits. Hence we delay the data u_1 by d_m cycles so that it appears as an input to the adder and subtracter circuits in the $(d_m + 1)$th cycle. This delay operation is performed with the help of a shift register $L_1, \ldots, L_{d_m+1}$ as shown in Fig. 5.2.

Observation 2: Every increment of j in Alg. 11 requires a new ω (line 17). If the multiplier has d_m pipeline stages, then the register-ω in Fig. 5.1 is updated with the new value of ω in the $(d_m + 2)$th cycle. Since this new ω is used by the next butterfly operations, the data dependency results in an interruption in the chain of butterfly operations for $d_m + 1$ cycles. In any m-loop, the total number of such *interruption cycles* is $(m/2 - 1) \cdot (d_m + 1)$.

To reduce the number of interruption cycles, we use a small look-up table to store a few twiddle factors. Let the look-up table (red dashed rectangle in Fig. 5.2) have l registers containing the twiddle factors $(\omega, \ldots \omega\omega_m^{l-1})$. This look-up table is used to provide the twiddle factors during the butterfly operations for say $j = j'$ to $j = j' + l - 1$. The next time j increments, new twiddle factors are required for the butterfly operations. We multiply the look-up table with ω_m^l to compute the next l twiddle factors $(\omega\omega_m^l, \ldots \omega\omega_m^{2l-1})$. The multiplications are independent of each other and hence can be processed in a pipeline. The butterfly operations are resumed after $\omega\omega_m^l$ is loaded in the look-up table. Thus using a small-look-up table of size l we reduce the number of interruption cycles to $(\frac{m}{2l} - 1) \cdot (d_m + 1)$. In our architecture we use $l = 4$; a larger value of l will reduce the number of interruption cycles, but will cost additional registers.

Optimal pipeline strategy for speed:

During the execution of any m-loop in Alg. 11, the number of butterfly operations is $n/2$. In the pipelined NTT-ALU, the cycle requirement for the $n/2$ butterfly operations is slightly larger than $n/2$ due to an initial overhead. The state machine jumps to the ω calculation state $\frac{m}{2l}-1$ times resulting in $(\frac{m}{2l}-1)\cdot(d_m+1)$ interruption cycles. Hence the total number of cycles spent in executing any m-loop can be approximated as shown below:

$$Cycles_m \approx \frac{n}{2} + (\frac{m}{2l} - 1) \cdot (d_m + 1) \ . \tag{5.2}$$

Assume that the delay of the critical path with no pipeline stages is D_{comb}. When the critical path is split into balanced-delay stages using pipelines, the resulting delay (D_s) can be approximated by $\frac{D_{comb}}{(d_m+d_a)}$, where d_m and d_a are the number of pipeline stages in the modular multiplier and the modular adder (subtracter) respectively. Since the delay of the modular adder is small compared to the modular multiplier, we have $d_a \ll d_m$. Now the computation time for

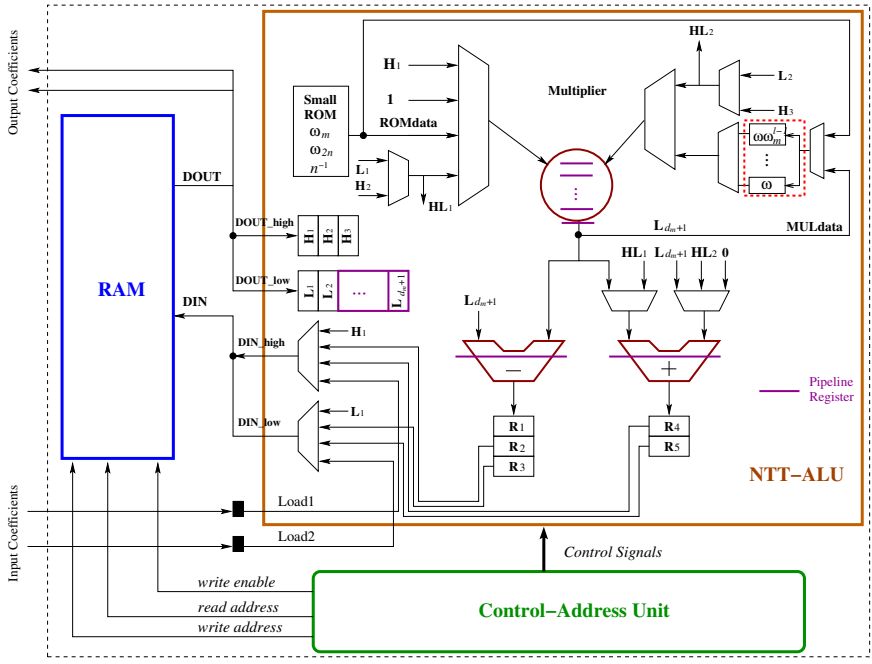

Figure 5.2: Pipelined Hardware Architecture for NTT

the m-loop is approximated as

$$T_m \approx \frac{D_{comb}}{(d_m + d_a)}\left[\frac{n}{2} + (\frac{m}{2l} - 1) \cdot (d_m + 1)\right] \approx D_s \frac{n}{2} + C_m \ . \tag{5.3}$$

Here C_m is constant (assuming $d_a \ll d_m$) for a fixed value of m. From the above equation we find that the minimum computation time can be achieved when D_s is minimum. Hence we pipeline the datapath to achieve minimum D_s. The DSP based coefficient multiplier is optimally pipelined using the Xilinx IPCore tool, while the modular reduction block is suitably pipelined by placing registers between the cascaded adder and subtracter circuits.

5.6 The ring-LWE encryption scheme

Pöppelmann et al. [106] optimized the computation cost of the `LPR` public key encryption scheme by keeping the fixed polynomials in the NTT domain. The message encryption and decryption operations require three and two NTT computations respectively. We reduce the number of NTT operations for decryption from two to *one*. The proposed ring-LWE encryption scheme is described below:

1. `LPR.KeyGen`(a) : Choose a polynomial $r_1 \in R_q$ from $\mathcal{X}_\sigma$, choose another polynomial r_2 with binary coefficients and then compute $p = r_1 - a \cdot r_2 \in R_q$. The NTT is performed on the three polynomials a, p and r_2 to generate $\tilde{a}$, $\tilde{p}$ and $\tilde{r_2}$. The public key is $(\tilde{a}, \tilde{p})$ and the private key is $\tilde{r_2}$.

2. `LPR.Encrypt`$(\tilde{a}, \tilde{p}, m)$: The message m is first encoded to $\bar{m} \in R_q$. Three polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext is then computed as:

$$\tilde{e_1} \leftarrow NTT(e_1); \quad \tilde{e_2} \leftarrow NTT(e_2)$$

$$(\tilde{c_1}, \tilde{c_2}) \leftarrow \left(\tilde{a} * \tilde{e_1} + \tilde{e_2};\ \tilde{p} * \tilde{e_1} + NTT(e_3 + \bar{m})\right) \ .$$

3. `LPR.Deccrypt`$(\tilde{c_1}, \tilde{c_2}, \tilde{r_2})$: Compute m' as $m' = INTT(\tilde{c_1} * \tilde{r_2} + \tilde{c_2}) \in R_q$ and recover the original message m from m' using a decoder.

The scheme requires both encryption and decryption to use a common primitive root of unity.

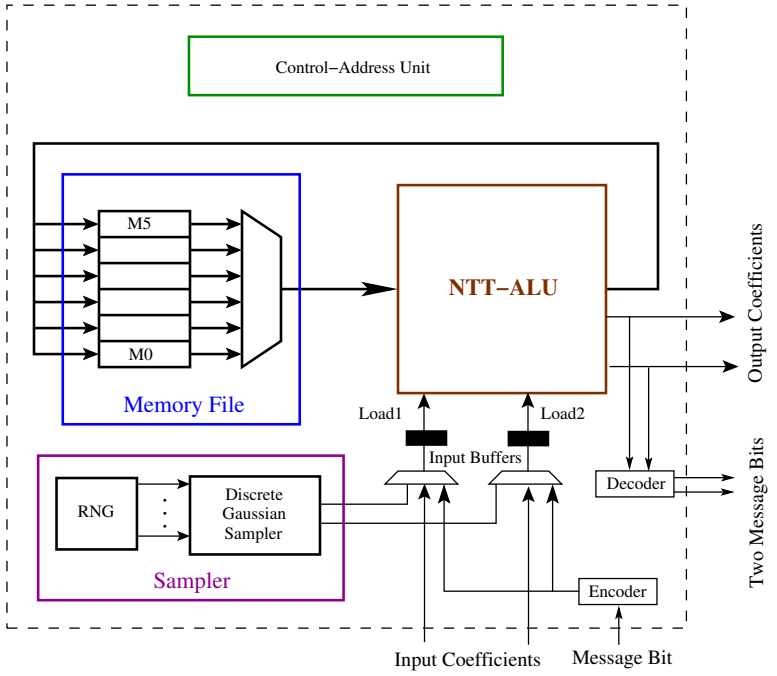

Figure 5.3: Ring-LWE Cryptoprocessor

5.6.1 Hardware architecture

Fig. 5.3 shows the hardware architecture for the ring-LWE encryption system. The basic building blocks used in the architecture are: the memory file, the arithmetic unit, the discrete Gaussian sampler and the control-address generation unit. The arithmetic unit is the NTT-ALU that we described in the previous section. Here we briefly describe the memory file and the discrete Gaussian sampler.

The Memory File is designed to support the maximum memory requirement that occurs during the encryption of the message. Six memory blocks M_0 to M_5 are available in the memory file and are used to store $\bar{a}$, $\bar{p}$, e_1, e_2, e_3 and $\bar{m}$ respectively. The memory blocks have width $2\lceil \log q \rceil$ bits and depth $n/2$. All six memory blocks share a common read and a write address and have a common data-input line, while their data-outputs are selected through a multiplexer. Any of the memory blocks in the memory file can be chosen for read and write operation. Due to the common addressing of the memory blocks, the memory file supports one read and one write operation in every cycle.

The Discrete Gaussian Sampler is based on the Knuth-Yao sampler architecture that we designed in the last chapter. The sampler does not include the shuffling countermeasure as in this work we intend to design the core of the public key encryption processor and measure its performance without the overhead of countermeasures. The sampler architecture has a sufficiently large precision and tail-bound to satisfy a maximum statistical distance of 2^{-90} to a true discrete Gaussian distribution for both $s = 11.32$ and $s = 12.18$. Two look-up tables are used to speedup the sampling operation. The first lookup table maps eight random bits and the second lookup table maps five random bits. When the second lookup operation fails (probability<0.0016) then bit-scan based Knuth-Yao random walk is started with the initial distance obtained from the second lookup operation.

The Cycle Count for the encryption and decryption operations can be minimized in the following way. During the encryption operation, first the three error polynomials e_1, e_2 and e_3 are generated by invoking the discrete Gaussian sampler $3n$ times. Next the encoded message $\bar{m}$ is added to e_3 and then three consecutive forward NTT operations are performed on e_1, e_2 and $(e_3 + \bar{m})$. Finally the ciphertext $\tilde{c_1}$, $\tilde{c_2}$ is obtained using two coefficient-wise multiplications followed by two polynomial additions and two rearrangement operations. The decryption operation requires one coefficient-wise multiplication, one polynomial addition and finally one inverse NTT operation.

During the encryption operation, $3n$ samples are generated to construct the three error polynomials. Our fast Knuth-Yao sampler architecture requires 805 and 1644 cycles for the dimensions 256 and 512 respectively on average to generate the three error polynomials. The polynomial addition and point-wise multiplication operations require n cycles each with a small overhead. The consecutive processing of I forward NTTs share a fixed computation cost fc_{fwd} and require in total $fc_{fwd} + I \times \frac{n}{2}\log(n)$ cycles. Similarly I consecutive inverse NTTs are processed in $fc_{inv} + I \times \frac{n}{2}\log(n) + I \times n$ cycles. One interesting point is that the fixed cost fc_{inv} is larger than fc_{fwd} as it includes the computation of ω_{2n}^{i}/N (Sect. 5.2) for $i = (0 \ldots n-1)$. This observation has been used to optimize the overall ring-LWE based encryption scheme in Sect. 5.6. The additional $I \times n$ cycles during the inverse NTTs are required to multiply the coefficients by the scaling factors. The rearrangement of polynomial coefficients after an NTT operation requires less than n cycles. From the above cycle counts for each primitive operations, we see that the encryption and decryption operations require total $fc_{fwd} + \frac{3}{2}n\log(n) + 10n$ and $fc_{inv} + \frac{n}{2}\log(n) + 3n$ cycles respectively along with additional overhead. Our ring-LWE architecture has the fixed computation costs $fc_{fwd} = 667$ and $fc_{inv} = 1048$ cycles for $n = 256$; and $fc_{fwd} = 1139$ and $fc_{inv} = 1959$ cycles for $n = 512$.

5.7 Experimental results

We have implemented the LPR ring-LWE cryptosystem on the Xilinx Virtex 6 FPGA for the parameter sets (n, q, s): (256,7681,11.32) and (512,12289,12.18). The area and performance results are obtained from the Xilinx ISE12.2 tool after place and route analysis and are shown in Table 5.1. In the table we also compare our results with other reported hardware implementations of the ring-LWE encryption scheme.

Our implementations are both fast and small due to the proposed computational optimizations and resource efficient design style. The cycle counts shown in the table do not include the cycles for data loading or reading operations. Our Knuth-Yao samplers have less than 2^{-90} statistical distances from the corresponding true discrete Gaussian distributions and consume around 164 LUTs and have delay less than $2.5ns$ (with optimization goal for speed). Such a small delay makes the sampler suitable for integration in the pipelined ring-LWE processor under a single clock domain. We use nine parallel true random bit generators [51, 34] to generate the random bits for the sampler. The set of true random bit generators consumes 378 LUTs and 9 FFs.

The first hardware implementation of the LPR ring-LWE encryption scheme in [52] uses a heavily parallel architecture to minimize the number of clock cycles for the NTT computation. Due to the many parallel computational blocks, the architecture is very large (0.29 million LUTs and 0.14 million FFs for $n = 256$) and does not even fit on the largest FPGA of the Virtex 6 family. Performance results such as cycle count and frequency are not reported in their paper. The architecture uses a Gaussian distributed array for sampling of the error coefficients up to a tail-bound of $\pm 2s$.

The implementation in [106] is small and fast due to its resource-efficient design style. A high operating frequency is achieved using pipelines in the architecture. The architecture uses a ROM that keeps all the twiddle factors required during the NTT operation. This approach reduces the fixed computation cost (fc) but consumes block RAM slices in FPGAs. Additionally, the parallel RAM blocks in the NTT processor result in a larger memory requirement compared to our design. The discrete Gaussian sampler is based on the inversion sampling method [33] and has a maximum statistical distance of 2^{-22} to a true discrete Gaussian distribution. Since the inversion sampling requires many random bits to output a sample value, an AES core is used as a pseudo-random number generator. The AES core itself consumes an additional 803 LUTs and 341 FFs compared to our true random number generator. Another reason behind the larger area consumption of [106] compared to our architecture is due to the fact that the architecture supports different parameter sets at synthesis time. Our

Table 5.1: Performance and Comparison

Implementation Algorithm	**Parameters**	**Device**	**LUTs/FFs/ DSPs/BRAM18**	**Freq (MHz)**	**Cycles/Time(μs)**	
					Encryption	**Decryption**
Our RLWE	(256,7681,11.32)	V6LX75T	1349/860/1/2	313	6.3k/20.1	2.8k/9.1
Our RLWE	(512,12289,12.18)		1536/953/1/3	278	13.3k/47.9	5.8k/21
RLWE [106]	(256,7681,11.32)	V6LX75T	4549/3624/1/12	262	6.8k/26.2	4.4k/16.8
RLWE	(512,12289,12.18)	V6LX75T	5595/4760/1/14	251	13.7k/54.8	8.8k/35.4
RLWE-Enc[105]	(256,4096,8.35)	S6LX9	317/238/95/1	144	136k/946	-
RLWE-Dec			112/87/32/1	189	-	66k/351
ECC[110]	Binary-233	V5LX85T	18097/-/5644/0	156	1.9k/12.3	1.9k/12.3
NTRU[63]	NTRU-251	XCV1600E	27292/5160/14352/0	62.3	-/1.54	-/1.41

ring-LWE processor is also designed to achieve scalability for various parameter sets. In our architecture the control block remains the same; while only the data-width and the modular reduction block changes for different parameter sets. Hence our architecture is also configurable by generating the HDL codes for various parameter sets using a C program.

Although our architecture does not use a dedicated ROM for storing the twiddle factors, it still achieves slightly smaller cycle count and faster computation time compared to [106]. The encryption scheme in [106] computes one forward and two inverse NTTs; while our encryption scheme computes only forward NTTs and hence does not require the $4n$ cycles for the scaling operation. Additionally our negative convolution method is free from the precomputation that takes n cycles in [106]. Hence we save $5n$ cycles in total during the NTT operations in an encryption operation. Since the fixed computation cost fc_{fwd} is smaller than $5n$, we gain in cycle count for the encryption operation. The decryption operation in our case is trivially faster than [106] as only one NTT is performed. We also reduce the area and memory requirement significantly compared to [52, 106]. This reduction is achieved by our resource-efficient design decisions such as 1) absence of a dedicated ROM for the twiddle factors, 2) an efficient RAM access and storage scheme, 3) use of one modular multiplier, 4) use of a smaller and faster (low-delay) discrete Gaussian sampler, and finally 5) the resource sharing between different computations.

The lightweight implementation [105] proposes ring-LWE encryption and decryption architectures targeting small area at the cost of performance. The implementation uses a quadratic-complexity multiplier instead of a complicated NTT based polynomial multiplier. Additionally the special modulus also saves some amount of area as the modular reduction is free of cost. However if we consider a similar quadratic-complexity multiplication based architecture in the dimension $n = 512$, then the cycle requirement will be nearly 40 times compared to our NTT-based ring-LWE processor. Our target was to use FPGA resources more efficiently without affecting the performance and to achieve similar speed as [106].

We also compare our results with other cryptosystems such as ECC and NTRU. The ECC processor [110] over the NIST recommended binary field $GF(2^{233})$ requires 12.3 μs to compute one scalar multiplication and is faster than our ring-LWE processor. However the ECC processor is designed to achieve high speed and hence consumes very large area compared to our ring-LWE processor. The NTRU scheme [63] is much faster than our ring-LWE processor due to its less complicated arithmetic. However the parameters chosen for the implementation in [63] have security around 64 bits [59]. Though secure parameter sets for the NTRU based encryption have been proposed in [57], no hardware implementation for the secure parameter sets is available in the literature.

5.8 Summary

In this chapter we analyzed the NTT based polynomial multiplication algorithm and proposed several optimizations to increase its computational efficiency and reduce storage requirement. We applied these optimization tricks to design a compact hardware architecture for polynomial arithmetic in the ring-LWE encryption scheme. We finally integrated the polynomial arithmetic unit with the compact Knuth-Yao discrete sampler from the last chapter and designed a compact and efficient ring-LWE public key encryption processor.

The design methodology and the optimizations make the cryptoprocessor architecture suitable for resource-constrained platforms. Although the chapter focuses on implementation of the ring-LWE based encryption system, we finally remark that the proposed optimization techniques for the NTT computation are applicable for other lattice based cryptosystems where similar polynomial multiplications are performed. In the next chapters, we will design processors for homomorphic encryption schemes. There we will show that the proposed optimizations could be very helpful in reducing the computation time.

Followup works. In joint works (coauthorship) with de Clercq et al. [31] and Liu et al. [81], we implemented the `LPR` ring-LWE public key encryption on 32-bit ARM and 8-Bit AVR processors respectively. The proposed optimizations in this chapter were adapted in [31] for the 32-bit processor architecture. For the parameter set $(n, q, s) = (256, 7681, 11.32)$ the software requires 121,166 cycles per encryption and 43,324 cycles per decryption. The encryption would be an order of magnitude faster than an implementation of the elliptic curve based ECIES encryption scheme (see Sect. 2.2) on a similar platform if the elliptic curve point multiplier of [32] is used. The software by Liu et al. shows that fast implementation of the ring-LWE encryption scheme is feasible on resource-constrained 8-bit AVR processors.

Side channel analysis of a new cryptographic constructions has always received interest from the research community. In joint works (coauthorship) with Reparaz et al. [116, 114, 115] we developed masking based countermeasures against differential power analysis attacks. Oder et al. [96] proposed practical ring-LWE based public key encryption that is protected against adaptive chosen-ciphertext attacks and equipped with countermeasures against side channel attacks. Park et al. [98] mounted an SPA attack combined with chosen ciphertext attack on the ring-LWE encryption. The attack exploits the computational variations during modular additions on 8-bit processors.

Chapter 6

Modular architecture for somewhat homomorphic function evaluation

CONTENT SOURCES:

The chapter is based on an extension of the following publication.

Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, and Ingrid Verbauwhede. Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation. In *International Workshop on Cryptographic Hardware and Embedded Systems CHES* (2015).

Contribution: Main author.

6.1 Introduction

Since the construction of the first fully homomorphic encryption (FHE) scheme by Gentry [46] in 2009, many researchers have developed more efficient schemes to improve the performance of FHE [21, 22, 30, 137, 43, 47, 49, 91]. Despite these major advances, the FHE schemes are too slow to be used in practical applications. Even somewhat homomorphic encryption (SHE) schemes, that can perform a limited number of operations on the encrypted data, are also very slow. Software implementations still require minutes or hours to evaluate even

rather simple functions. For instance, evaluating the decryption of a lightweight block cipher SIMON-64/128 (block/key size 64/128 bits) [15], requires 4193 s (an hour and 10 minutes) on a 4-core Intel

Hardware accelerators have been successfully used for accelerating performance-critical computations in cryptology (see, e.g, [71]). When we started this research, only a few publications [142, 37, 90, 143] reported results on hardware-based acceleration of FHE and SHE. Moreover, the homomorphic schemes used in these implementations are not based on the ring-LWE problem. In this chapter we design a hardware accelerator to speedup the ring-LWE based somewhat homomorphic encryption scheme `YASHE`.

The chapter is structured as follows. Sect. 6.2 describes the system setup and the parameter set that we use. The next section contains a high level description of known optimization techniques to speed-up computations in modular polynomial rings and describes how we represent polynomials using the Chinese Remainder Theorem (CRT) in order to parallelize computations. We present our hardware architecture for YASHE in Sect. 6.4. Sect. 6.5 shows the performance results and the final section draws the summary.

6.2 System setup

As described in Sect. 2.4.2, the YASHE scheme computes in a polynomial ring of the form $R = \mathbb{Z}[x]/(f(x))$ where $f(x)$ is a monic irreducible polynomial of degree n. We choose the plaintext modulus $t = 2$, i.e., we evaluate bit-level operations. We put no restriction on $f(x)$, which allows us to deal with any cyclotomic polynomial $\Phi_d(x)$ and thus to utilize single instruction multiple data (SIMD) operations [127, 128]. Indeed to exploit the SIMD feature, we choose an irreducible polynomial $f(x)$ such that $f(x) \bmod 2$ splits into many different irreducible factors, each factor corresponding to "one slot" in the SIMD representation. It is easy to see that this excludes $f(x) = x^n + 1$ with n a power of two, since it results in only one irreducible factor modulo 2. Note that in Chap. 5 we took $f(x) = x^n + 1$ to compute polynomial multiplications without performing reductions modulo $f(x)$. With our current choice, we achieve SIMD, but we pay in modular reductions by $f(x)$.

We use the parameter set Set-III from [77] that supports homomorphic evaluations of SIMON-64/128; in particular $d = 65535$ (and thus the degree of $f(x)$ is $32768 = 2^{15}$), $\log_2(q) = 1228$ and χ_{err} a discrete Gaussian distribution with parameter $\sigma = 8$. When we started this research, the parameter set was presumed to have a 128-bit security level for the YASHE scheme [77]. The irreducible polynomial $f(x)$ splits modulo 2 in 2048 different irreducible

polynomials, which implies that we can work on 2048 bits in parallel using the SIMD method first outlined in [127].

6.3 High-level optimizations

To efficiently implement YASHE we have to analyze the two main operations in detail, namely homomorphic addition and homomorphic multiplication. Homomorphic addition is easy to deal with since this simply corresponds to polynomial addition in R_q. Homomorphic multiplication is much more involved and is the main focus of this chapter. As can be seen from the definition of YASHE.Mult in Sect. 2.4.2, to multiply two ciphertexts c_1 and c_2 one first needs to compute $c_1 \cdot c_2$ over the integers, then scale by t/q and round, before mapping back into the ring R_q. The fact that one first has to compute the result over the integers (to allow for the scaling and rounding) has a major influence on how elements of R_q are represented and on how the multiplication has to be computed.

Since each element in R_q is a polynomial of degree $n-1$, the result of a polynomial multiplication (without reduction modulo $f(x)$) will have degree $2n-2$. As such we choose the smallest $N = 2^k > 2n-2$, and compute the product of the two polynomials in the ring $\mathbb{Z}_q[x]/(x^N-1)$ by applying the N-point NTT (see Alg. 10). The NTT requires the N-th roots of unity to exist in $\mathbb{Z}_q$, so we either choose q a prime with $q \equiv 1 \bmod N$ or q a product of small primes q_i with each $q_i \equiv 1 \bmod N$. It is the latter choice that will be used throughout this work. The product of two elements $a, b \in R_q$ is then computed in two steps: firstly, the product modulo $x^N - 1$ (note that there will be no reduction, since the degree of the product is small enough) is computed using two NTT's, N pointwise multiplications modulo q and then finally, one inverse NTT. To recover the result in R_q, we need a reduction modulo $f(x)$. For this purpose, we use the Newton iteration method that we described in Sect. 2.5.2.

Note that the multiplication of c_1 and c_2 in YASHE.Mult is performed over integers. To get the benefit of NTT based polynomial multiplication, we perform this multiplication in a ring R_Q where Q is a sufficiently large modulus of size $\sim 2 \log q$ such that the coefficients of the result polynomial are in $\mathbb{Z}$.

CRT representation of polynomials:

The biggest challenge while designing a homomorphic processor is the complexity of computation. During a homomorphic operation, computations are performed on polynomials of degree 2^{15} or 2^{16} and coefficients of size ∼1,200 or ∼2,500 bits.

If we use a bit-parallel coefficient multiplier, then a $2,500 \times 2,500$-bit multiplier will not only result in an enormous area. On the other side, a word-serial multiplier is too slow for homomorphic computations.

To tackle the problem of long integer arithmetic, we take inspiration from the application of the CRT in the RSA cryptosystems. We choose the moduli q and Q as products of many small prime moduli q_i, such that $q = \prod_0^{l-1} q_i$ and $Q = \prod_0^{L-1} q_i$, where $l < L$. We map any long integer operation modulo q or Q into small computations moduli q_i, and apply CRT whenever a reverse mapping is required. We use the term *small residue* to represent coefficients modulo q_i and the term *large residue* to represent coefficients modulo q or Q.

YASHE.Mult in residue domain

Let us take two input ciphertext polynomials c_1 and $c_2 \in R_q$. The homomorphic multiplication steps are described below.

1. $\mathsf{Lift}_{q \to Q}$: Lift c_1 and c_2 to R_Q from R_q, i.e., compute the additional residue polynomials moduli q_j for $j \in [l, L-1]$. Note that c_1 and c_2 are represented as residue polynomials moduli q_i for $i \in [0, l-1]$ in R_q. So, first compute the coefficients modulo q in $(-q/2, q/2)$ by applying the CRT, and then compute the additional residue polynomials.

2. $\mathsf{PolyMultiply}_Q$: Compute the product polynomial $c = c_1 \cdot c_2 \in R_Q$ by computing multiplications of the residue polynomials moduli q_j for $j \in [0, L-1]$.

3. $\mathsf{Lift}_{Q \to q}$: Apply the CRT on the residue polynomials of c and compute the coefficients modulo Q in $(-Q/2, Q/2)$. Now compute the division-and-rounding operation to $c' = \lfloor \frac{t \cdot c}{q} \rceil$. Next, reduce the coefficients of c' modulo q in $(-q/2, q/2)$.

4. `WordDecomp` : Split the coefficients of c' into w-bit words to get the vector $\mathbf{c'}$ of $\lceil \lg q/w \rceil$ polynomials.

5. YASHE.KeySwitch : Compute the residue polynomials for each member of $\mathbf{c'}$ and then compute $c_{mult} = \langle \mathbf{c'}, \mathbf{evk} \rangle \in R_q$. This gives the result of the homomorphic multiplication as a set of residue polynomials in R_q.

The polynomial arithmetic on the residue polynomials can be performed in parallel. The size of the moduli q_i is an important design decision and depends on the underlying platform. We implement the hardware accelerator on the

Xilinx ML605 board, which has a Virtex-6 FPGA. The FPGA provides 24×17-bit unsigned DSP multipliers to perform integer multiplications. We could implement a slightly larger integer multiplier by combining a DSP multiplier with LUT-based logic. In this work we choose 30-bit prime q_i that satisfy $q_i \equiv 1 \mod N$. The reasons for selecting only 30-bit of primes are: 1) there are sufficiently many primes of size 30-bit to compose 1,228-bit q and 2,471-bit Q, 2) the data-paths for performing computations modulo q_i become symmetric, and 3) the basic computation blocks, such as adders and multipliers of size 30-bit can be implemented efficiently using the available DSP slices and a few LUTs.

6.4 Architecture

In this section we design a hardware architecture, which we call HE-coprocessor, to accelerate YASHE.Add and YASHE.Mult . The design decisions take account of the computational resources available on the Xilinx ML605 board which has a medium size Xilinx Virtex-6 FPGA XC6VLX240T. Since the ciphertexts are large, of size 4.8MB, we use the DDR3 memory of the board to store the ciphertexts. During a computation, portions of the ciphertext(s) are read from the DDR memory and stored in the on-FPGA BRAMs. After the computation, the result is written back in the DDR memory. The speed of the communication between the DDR memory and the FPGA has a major impact on the performance. In this work we restrict the data-size to 256 bits per DDR memory access.

In the remaining part of this section, we describe our design decisions and optimization tricks.

6.4.1 Architecture for polynomial arithmetic

From Sect. 6.3 we see that YASHE.Mult involves arithmetic on polynomial with large degree and large coefficient size. Following the design methodology of the public key encryption processor of Chap. 5, we apply the NTT to compute the multiplications of the residue polynomials. However, a simple scale-up of the NTT computation core of Chap. 5 will not be enough as it will take more than 524K cycles to compute an N-point NTT. Note that the NTT (Alg. 10 in Chap. 5) is amicable to parallelism. Hence on FPGAs, we use parallel butterfly cores to reduce the number of cycles.

Optimization in the routing:

Let a residue polynomial of N coefficients be stored in b BRAMs and then processed using v butterfly cores. For simplicity, let v be a divisor of b and a power of two. We can split the NTT computation into equal parts among the v parallel cores. However there are two main technical issues related to the memory access that would affect the performance of the NTT computation. The first one is: all the parallel cores access the BRAMs simultaneously. Since a simple dual port BRAM has one port for reading and one port for writing, it can support only one read and write in a clock cycle. This puts the restriction that a memory block can be read (or written) by one butterfly core in a cycle, i.e., the generation of the BRAM-addresses by the parallel cores should be free from conflicts.

The second issue is related to the routing complexity. A residue polynomial is stored in many BRAMs, and hence, if a core needs to access a BRAM that is very far from it, then the routing of wires will very long. Note that in the basic NTT (Alg. 10) we see that the maximum difference between the indexes of the two coefficients is $N/2$. For the chosen parameter set $N = 2^{16}$; hence fetching data from memory locations at a relative distance of 2^{15} will result in a very long routing, and thus could drastically reduce the clock frequency.

To address these two technical issues, we have developed a memory access scheme by analyzing the generation of indexes during NTT computation. We segment the set of b BRAMs into b/v groups. The read ports of a group are accessed by only one butterfly core. This *dedicated read* prevents any sort of conflict during the memory read operations. Moreover, in the FPGA the group of BRAMs can be placed close to the corresponding butterfly core and thus the routing complexity can be reduced.

We describe the proposed memory access scheme during an execution of the NTT by parallel cores in Alg. 12. Following the memory-efficient NTT algorithm of Chap. 5, the module *butterfly-core* in Alg. 12 performs butterfly operations on two coefficent pairs. In the algorithm the v parallel butterfly cores of a processor are indexed by c where $c \in [0, v-1]$. During the m-th loop of a NTT, the twiddle factor in the c-th core is initialized to a constant value $\omega_{m,c}$. In the hardware, these constants are stored in a ROM. The counter $I_{twiddle}$ denotes the interval between two consecutive calculations of the twiddle factors. Whenever the number of butterfly operations ($N_{butterfly}$) becomes a multiple of $I_{twiddle}$, a new twiddle factor is computed. The c-th butterfly core reads the $c-$th group of BRAMs $MEMORY_c$ using two addresses $address_1$ and $address_2$. The addresses are computed from the counters: *base*, *increment*, and *offset*, that represent the starting memory address, the increment value, and

the difference between $address_1$ and $address_2$ respectively. A butterfly core outputs the two addresses and the four coefficients $s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}$. These output signals from the parallel butterfly cores are collected by a set of parallel modules *memory-write* that are responsible for writing the groups of BRAMs. The input coefficients that will be read by the adjacent butterfly core in the next iteration of the m-th loop, are selected for the writing operation in $MEMORY_c$ by the c-th memory-write module. The top module *Parallel-NTT* instantiates v butterfly cores and memory write blocks. These instances run in parallel and exchange signals.

Internal architecture of the PAU:

In Fig. 6.1 we show the internal architecture of the cores that we use to perform arithmetic on the residue polynomials. The cores have been designed following the footprints of the polynomial arithmetic core of Chap. 5. The input register bank contains registers to store data from the BRAMs. In addition, the register bank also contains shift registers to delay the input coefficients in a pipeline during a NTT computation. The register bank has several ports to provide data to several other components present in the core. We use the common name $D_{regbank}$ to represent all data-outputs from the register bank. The small ROM block in Fig. 6.1 contains the twiddle factors and the value of N^{-1} to support the computation of NTT and INTT.

The integer multiplier (shown as a circle in Fig. 6.1) is a 30×30-bit multiplier. We maintain a balance between area and speed by combining two DSP multipliers and additional LUT based small multipliers to form this multiplier. After an integer multiplication, the result is reduced using the Barrett reduction circuit shown in Fig. 6.1. We use the Barrett reduction technique due to two reasons. The first reason is that the primes used in this implementation are not of pseudo-Mersenne type which support fast modular reduction technique [54]. The second reason is that the cores are shared by all the prime moduli, and hence, a generic reduction circuit is more preferable than several dedicated reduction circuits. The Barrett reduction circuit is bit parallel to process the outputs from the bit-parallel multiplier in a flow. The reduction consists of three 31×31-bit multipliers and additional adders and subtractors. The multipliers are implemented by combining two DSP multipliers with additional LUTs. Thus in total, the Barrett reduction block consumes six DSP multipliers. Beside performing the modular reduction operations, the multipliers present in the Barrett reduction circuit can be reused to perform 30×59-bit multiplications during the CRT computations.

The adder/subtracter circuits after the Barrett reduction block in Fig. 6.1 are

Algorithm 12: Routing Efficient Parallel NTT using v cores

```
/* This module computes butterfly operations                                          */
module butterfly-core(input c; output m, address_1, address_2, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c})
begin
    (I_twiddle, offset) ← (N/2, 1)
    for m = 0 to log N − 1 do
        ω_m ← 2^m-th primitiveroot(1)
        N_butterfly ← 0                 /* Counts the number of butterfly operation in a m-loop */
        ω ← ω_{m,c}                     /* Initialization to a power of ω_m for a core-index c */
        for base = 0 to base < offset do
            increment ← 0
            while base + offset + increment < N/(2v) do
                (address_1, address_2) ← (base + increment, base + offset + increment)
                (t_1, u_1) ← MEMORY_c[address_1]          /* Read from c-th group of RAMs */
                (t_2, u_2) ← MEMORY_c[address_2]
                if m < log N − 1 then
                    (t_1, t_2) ← (ω · t_1, ω · t_2)
                    (s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}) ← (u_1 + t_1, u_1 − t_1, u_2 + t_2, u_2 − t_2)
                    N_butterfly ← N_butterfly + 2
                    increment = increment + 2 · offset
                    if N_butterfly ≡ I_twiddle then ω ← ω · ω_m^{v/2}
                end
                else
                    t_1 ← ω · t_1; ω ← ω · ω_m^{v/2}
                    t_2 ← ω · t_2; ω ← ω · ω_m^{v/2}
                    (s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}) ← (u_1 + t_1, u_1 − t_1, u_2 + t_2, u_2 − t_2)
                    N_butterfly ← N_butterfly + 2
                    increment = increment + 2 · offset
                end
            end
        end
        I_twiddle ← I_twiddle/2
        if offset < v/2 then offset ← 2 · offset
    end
end
/* This module writes the coefficients computed by two butterfly-cores                */
module memory-write(input c, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
begin
    if 2^m < v/2 then gap ← 2^m
    else gap ← v/2                  /* This represents the index gap between the two cores */
    if c < v/2 then
        MEMORY_c[address_1] ← (s_{2,c}, s_{1,c})
        MEMORY_c[address_2] ← (s_{2,c+gap}, s_{1,c+gap})
    end
    else
        MEMORY_c[address_1] ← (s_{4,c}, s_{3,c})
        MEMORY_c[address_2] ← (s_{4,c+gap}, s_{3,c+gap})
    end
end
/* This is the top module that executes butterfly-core in parallel                    */
module Parallel-NTT()
begin
    butterfly-core bc_0(0, m, address_1, address_2, s_{1,0}, s_{2,0}, s_{3,0}, s_{4,0})
    memory-write mw_0(0, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
    ...
    butterfly-core bc_{v−1}(v − 1, m, address_1, address_2, s_{1,v−1}, s_{2,v−1}, s_{3,v−1}, s_{4,v−1})
    memory-write mw_{v−1}(v − 1, m, address_1, address_2, s_{1,0}, ... s_{4,v−1})
end
```

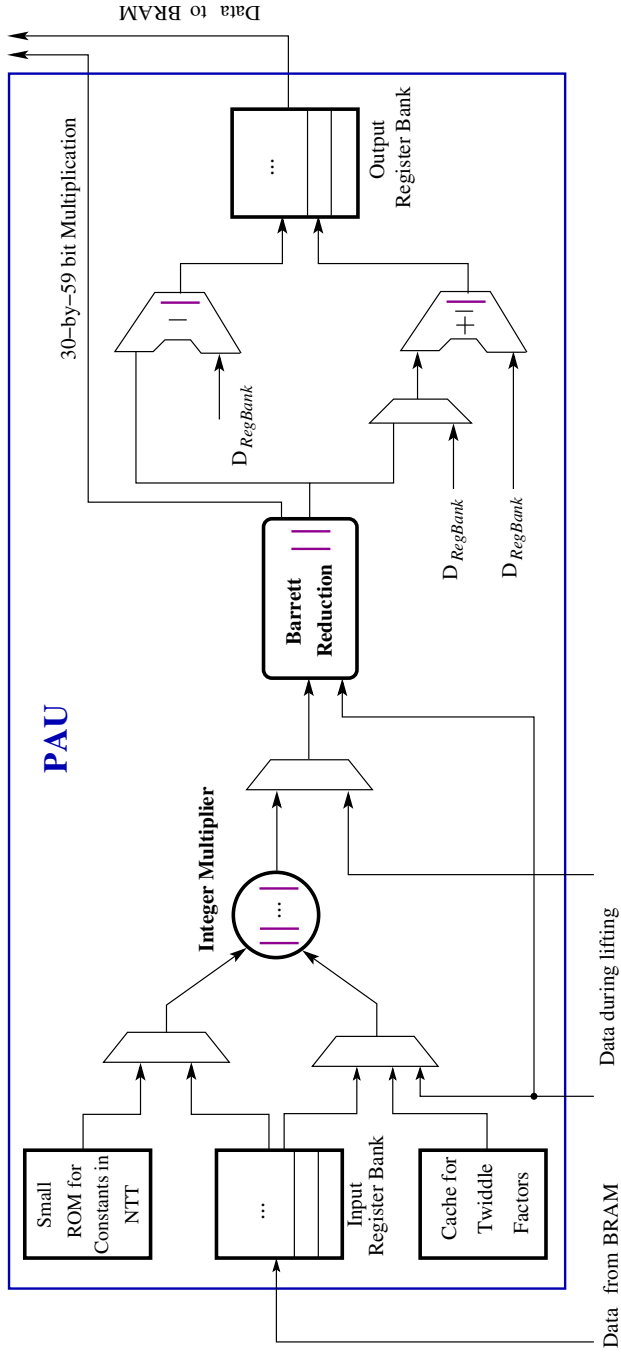

Figure 6.1: Architecture for the Vertical Cores

Algorithm 13: Calculation of the reverse of an index. A polynomial of 2^{16} coefficients is stored as 2^{15} coefficient pairs in 16 memory elements. BiasTable contains 16 bias values {0, 16, 8192, 8208, 4096, 4112, 12288, 12304, 2048, 2064, 10240, 10256, 6144, 6160, 14336, 14352} corresponding to the 16 memory elements.

Input: An index of a coefficient pair where $index \in [0, 2^{15} - 1]$
Output: Reverse of the index $reverse\_index \in [0, 2^{15} - 1]$

```
1  begin
2      c ← index ≫ 11 ;                          /* index of the memory element */
3      low ← index&31 ;                          /* least five bits */
4      high ← (i ≫ 5)&63 ;                       /* next six bits */
5      lsb ← low[0] ;
6      low ← low − lsb ;
7      low_reverse ← bitreverse(low) ;           /* bits in reverse order */
8      high_reverse ← bitreverse(high) ;
9      bias ← BiasTable[c] ;
10     reverse_index ← bias + (high_reverse ≪ 5) + low_reverse + (lsb ≪ 14) ;
11 end
```

used to compute the butterfly operations during a NTT computation and to perform coefficient-wise additions and subtractions of polynomials. Finally, the results of a computation are stored in the output register bank and then the registers are written back in the memory. To achieve high operating frequency, we put pipeline registers (shown as magenta colored lines) in the data paths of the computation circuits.

External memory access during NTT:

During an NTT, the coefficients of the residue polynomial are read sequentially from the DDR memory and then loaded in the 16 internal memory blocks. For this purpose the 256-bit DDR3 interface is used to receive four coefficient pairs (i.e. eight coefficients) in a burst. However Alg. 12 generates the output coefficient pairs in a permutation that is different from their initial arrangement. The coefficients pairs are written back in the DDR memory in the right arrangement using Alg. 13. For a write address *index*, the coefficient pair from the address *reverse_index* should be read from the internal memory. Note that we perform this rearrangement of the coefficients after the completion of an NTT following Alg. 12; whereas in the traditional NTT Alg. 10 this rearrangement is performed in the beginning using the *bitreverse* function.

6.4.2 Architecture for lifting back and forth in $R_q \leftrightarrow R_Q$

In Sect. 6.3 we described the lifting operations that we need to perform during YASHE.Mult. In this section we describe the computational steps that we follow to implement the lifting operations, and then we describe the hardware architectures of the building blocks. In the end we design a unified architecture for computing the two lifting operations.

Computation steps for $\text{Lift}_{q\rightarrow Q}$

Let for an integer $a \mod q$, the residues be $[a]_{q_i}$ for $i \in [0, l-1]$. So we are interested in computing $[a]_{q_j}$ for $j \in [l, L-1]$. We first compute the sum of products for $i \in [0, l-1]$ as follows.

$$a_{sp} = \sum [a]_{q_i} \cdot (\frac{q}{q_i}) \cdot [(\frac{q}{q_i})^{-1}]_{q_i} = \sum [a]_{q_i} \cdot b_i \ . \tag{6.1}$$

Next we compute $[a']_{q_j}$ for $j \in [l, L-1]$ using

$$[a']_{q_j} \quad = \quad \big[\sum [a]_{q_i} \cdot [b_i]_{q_j} \big]_{q_j} \ . \tag{6.2}$$

Note that $[b_i]_{q_j}$ are 30-bit integers. Finally, we compute the residues $[a]_{q_j}$ for $j \in [l, L-1]$ using the following equation:

$$[a]_{q_j} = \big[[a']_{q_j} - [\lfloor a_{sp}/q \rfloor]_{q_j} \cdot [q]_{q_j} - sign \cdot [q]_{q_j} \big]_{q_j} \ . \tag{6.3}$$

This computation involves a division of a_{sp} by q. The $sign$ takes a value 0 or 1 depending on $a_{sp} - \lfloor a_{sp}/q \rfloor \cdot q$ is smaller than $q/2$ or not.

Computation steps for $\text{Lift}_{Q\rightarrow q}$

We compute the sum of products a_{sp} from the residue polynomials moduli q_j for $j \in [0, L-1]$.

$$a_{sp} = \sum [a]_{q_j} \cdot [(\frac{Q}{q_j})^{-1}]_{q_j} \cdot (\frac{Q}{q_j}) = \sum [a']_{q_j} \cdot b_j \ . \tag{6.4}$$

Here the values $[(\frac{Q}{q_j})^{-1}]_{q_j}$ are 30-bit integers and hence the computation $[a']_{q_j} = [a]_{q_j} \cdot [(\frac{Q}{q_j})^{-1}]_{q_j}$ is a 30-bit modular multiplication. Next we reduce a_{sp} by Q and

get a_Q in $(-Q/2, Q/2)$. Then the division and rounding operation is performed on a_Q and the result is reduced modulo q to a value in $(-q/2, q/2)$.

Unified architecture

Note that $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$ operations involve similar computation steps such as sum of products in Eq. 6.1, 6.2 and 6.4, and divisions by q. Hence we design a unified architecture to compute both $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$. The architecture is composed of four blocks: 1) sum of products, 2) reduction modulo Q, 3) division-and-rounding, and 4) reduction modulo q. The blocks are described as follows.

sum of products block. Fig. 6.6 shows a multiply-and-accumulate (MAC) core to compute the sum of products in Eq. 6.1, 6.2 and 6.4. In the figure, the 'multiplier' block is borrowed from the PAU (Fig. 6.1). Since there are 16 PAU cores in the HE-processor, we instantiate 16 MAC cores. These cores are divided into two parallel MAC-groups: MAC-0 to MAC-7 form the first group, and MAC-8 to MAC-15 form the second group. Each MAC-group is responsible for computing one sum of products.

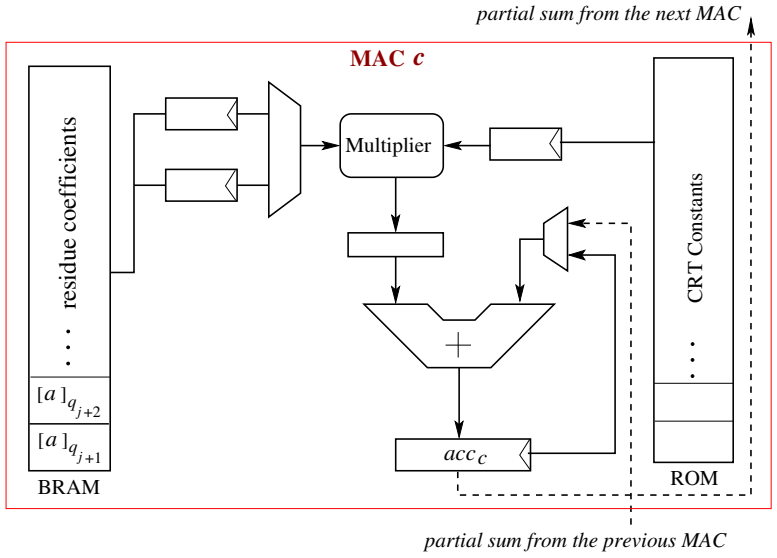

Figure 6.2: Architecture for computing sum of products

The ROM block in the MAC core is a loadable memory and is used to store the constants for $\mathsf{Lift}_{q\to Q}$ or $\mathsf{Lift}_{Q\to q}$. We set the word size of the ROM to 59 bits.

Note that Eq. 6.1 and 6.4 require multiplications of 30-bit coefficients $[a]_{q_i}$ by long b_i. The MAC cores compute these long multiplications word-serially using the 31×59-bit integer multipliers that are present inside the multiplier blocks. Alg. 14 shows the word-serial computation of the sum of products by the c-th MAC core. In the algorithm we assume that the MAC core is responsible for the accumulation of m products and each b_i has w 59-bit words in the ROM. The k-loop computes the kth 59-bit word of the partial result in sum_c. In this way each MAC core computes a partial sum of products.

Now the partial results from the MAC cores are accumulated to get the kth word of the final sum of products. The sequence of accumulation in the first group happens as follows: sum_0 from MAC-0 is forwarded to MAC-1. Now MAC-1 computes Alg. 14 with the initialization of acc_1 to sum_0 and computes sum_1. Following the same sequence, MAC-3 computes sum_3. In parallel to this flow, MAC-7 downto MAC-4 computes sum_4. Finally sum_4 is added with sum_3 in MAC-3 to get the word of the final sum of products.

Algorithm 14: Calculation of partial sum in Eq. 6.1

Input: Residue coefficients $[a]_{q_j}$, constants b_j, and MAC core index c
Output: Partially accumulated sum of products

```
1  begin
2      acc_c ← 0 ;
3      for k = 0 to w − 1 do
4          for j = 0 to m − 1 do
5              acc_c ← acc_c + [a]_{q_j} · b_j[k] ;
6          end
7          sum_c ← least59bits(acc_c) ;
8          acc_c ← acc_c ≫ 59 ;
9      end
10     sum_c ← least59bits(acc_c) ;
11 end
```

The computation of Eq. 6.2 requires sum of modular multiplication results. 30-bit coefficients $[a]_{q_i}$ are fetched from the BRAM and then multiplied with the 30-bit constants $[b_i]_{q_j}$ using the modular multiplier. Each MAC core computes a partial sum and then the partial sums are accumulated together. The final result is larger than 30-bits and is reduced modulo q_j during the computation of Eq. 6.3. In Fig. 6.3 we show the timing diagram for the pipeline processing.

Reduction modulo Q block Let a_{sp} be the sum of the products in Eq. 6.4. For the chosen parameter set, a_{sp} is 7 bits larger than Q. We first reduce a_{sp} to a value in $[0, Q-1]$ and then central-lift the result to a value in the range

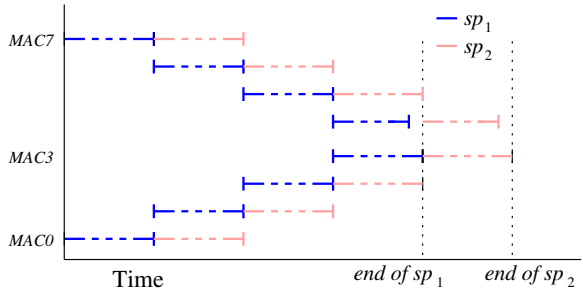

Figure 6.3: Timing diagram for pipeline processing of two consecutive sum-of-products (sp) by the first MAC-group.

$(-Q/2, Q/2)$. To reduce a_{sp} in $[0, Q-1]$, we sequentially reduce the extra bits of a_{sp} from the most significant side. The steps are shown in Alg. 15.

Algorithm 15: Reduction modulo Q

Input: An integer a_{sp} that is 7-bit larger than Q
Output: Integer a_{sp} in $[0, Q-1]$

```
1  begin
2      Q_n ← (Q ≪ 7) ;
3      for i = 0 to 6 do
4          if msb(a_sp) > 0 then
5              temp ← a_sp − Q_n ;
6              if temp ≥ 0 then
7                  a_sp ← temp ;
8              end
9          end
10         a_sp ← (a_sp ≪ 1) ;
11     end
12     a_sp ← (a_sp ≫ 7) ;
13     if a_sp ≥ Q then
14         a_sp ← a_sp − Q ;
15     end
16 end
```

A word-serial architecture for computing the reduction modulo Q is shown in Fig. 6.4. The architecture has three addressable memory components: M and M_t are used to store the computational data and M_Q is used to keep the modulus Q. These memory components are distributed RAMs of word size 59-bits and

depth 64. At the beginning of a computation, the input number a_{sp} is loaded in M. Then within the *for*-loop of Alg. 15, the words of a_{sp} are left-shifted by one position and then stored in M_t. Note that, line 5 has a conditional subtraction operation. In our implementation, the subtraction is performed word-serially using the subtraction circuit, and the result words are left-shifted by one position and then stored in M. Based on the sign of the subtraction, either M or M_t is used as the source of a_{sp} for the next computations.

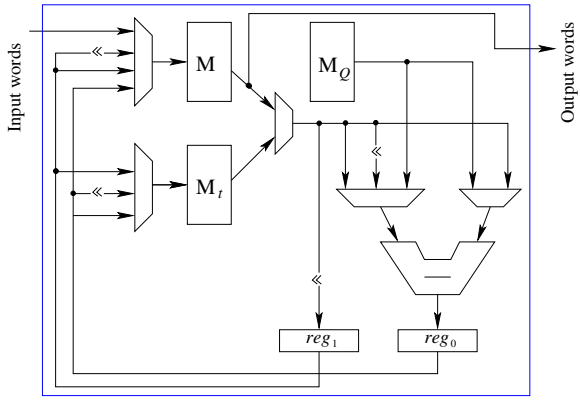

Figure 6.4: Architecture for reduction modulo Q

Division and Rounding Unit (DRU)

The DRU computes $\lfloor tc/q \rceil$ during $\mathsf{Lift}_{Q \to q}$ where $t = 2$, c is a coefficient computed from the reduction modulo Q, and $\lfloor \cdot \rceil$ denotes rounding towards the nearest integer. The division is carried out by precomputing the reciprocal $r = 2/q$ and then computing $r \times c$. The word size of the DRU was selected to be 118 bits (2×59) as a compromise between area and latency.

To round a division of two k-bit integers correctly to k-bits, the quotient must be computed correctly to $2k + 1$ bits [66, Theorem 5.2]. In our case, the computation of $\lfloor tc/q \rceil$ requires a division of a k_1-bit dividend by a k_2-bit divisor. The precision that we will need in this case to guarantee correct rounding, based on the above, is $k_1 + k_2 + 1$ bits. The divisor q is a 1228-bit constant integer and the dividend c is an at most 2470-bit integer, which gives a bound of 3699 bits. Hence, the reciprocal r is computed up to a precision of 32 118-bit words, of which 22 words are nonzero.

Fig. 6.5 shows the architecture of the DRU. The multiplication $r \times c$ is computed by using a 118×118-bit multiplier that computes $22^2 = 484$ partial multiplications. This multiplier performs a 118-bit Karatsuba multiplication by using three 59×59-bit multipliers generated with the Xilinx IP Core tool (which

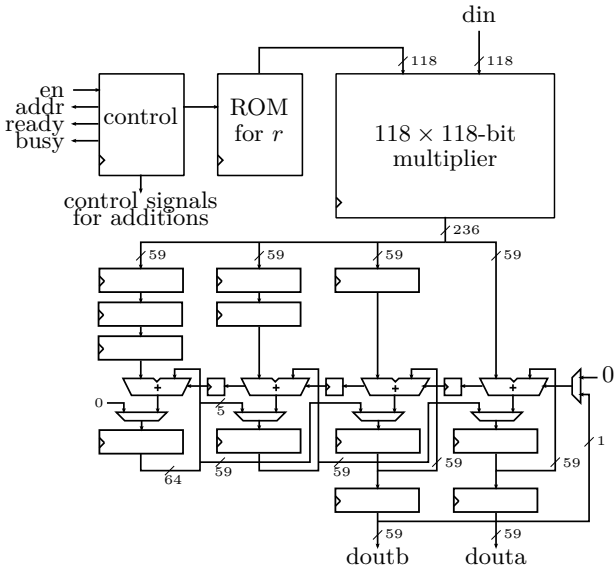

Figure 6.5: The Division and Rounding Unit (DRU)

supports only up to 64-bit multipliers). The 59-bit multipliers each require 16 DSP blocks giving the total number of 48 DSP blocks. In order to achieve a high clock frequency, the 118-bit multiplier utilizes a 23-stage pipeline, of which 18 stages are in the 59-bit multipliers (the optimal number according to the tool).

The partial products from the 118-bit multiplier are accumulated into a 241-bit $(2 \times 118 + 5)$ register using the Comba Alg [27]. These additions are computed in a 4-stage pipeline with three 59-bit adders and one 64-bit adder, which are all implemented with LUTs. Whenever all partial products of an output word have been computed, the register is shifted to the right by 118 bits and the overflowing bits are given at the output of the DRU. Once the computation proceeds to the first word after the fractional point, then the msb of the fractional part is added to the register in order to perform the rounding. The DRU has a constant latency of 687 clock cycles per coefficient.

The DRU is reused for computing $\lfloor a_{sp}/q \rfloor$ during the $\mathsf{Lift}_{q \to Q}$. The computation proceeds analogously to the above. The differences are that the reciprocal is now $r = 1/q$ and it needs to be computed only to a precision of 2493 bits (12 nonzero words) because c can be only 36 bits longer than q. The computation has a latency of 246 clock cycles per division.

Reduction modulo q block The architecture for this block is same as the architecture for the reduction modulo Q block; only the operands are smaller.

Note that in the first MAC-group, MAC-3 computes the final sum of products. So the reduction and division blocks are attached to the MAC-3 core. In Fig. 6.6 we show the connection of MAC-3 core with the remaining three blocks. Similarly in the second MAC-group, MAC-11 core is accompanied by the reduction and division blocks. The four blocks are in a pipeline during $\mathsf{Lift}_{Q\to q}$ to achieve optimum computation time. The division block takes the maximum cycles and hence determines the throughput of the entire pipeline. Every block contains additional memory elements to enable the pipeline processing: while one memory element is read by the next block in the pipeline, the other memory element is used to store the new results.

During $\mathsf{Lift}_{q\to Q}$ operation, the sum of products a_{sp} in Eq. 6.1 is computed by a MAC-group, and then it is passed to the DRU for the computation of $\lfloor a_{sp}/q \rfloor$. In parallel to this division, the MAC-group computes $[a']_{q_j}$ in Eq. 6.2. For the computation of Eq. 6.3, a small computation block (consisting of a multiplier, subtracter and some small memory components) is used in the pipeline. The *sign* is computed by performing arithmetic on the most significant words of a_{sp} and q. This block is common to both the MAC-groups as the amount of computation in Eq. 6.3 is small. The throughput of the pipeline during $\mathsf{Lift}_{q\to Q}$ is determined by the 'computation of a_{sp} followed by the division $\lfloor a_{sp}/q \rfloor$'.

External memory access:

The DDR memory access during the $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$ is more complicated than the memory access during NTT. Here we need to fetch the residue coefficients for different moduli, whereas during an NTT we fetch coefficients from a single moduli. So we design a customized DDR memory access interface for the lifting operations. Since the DDR-burst data length is 256 bits, at a time we read eight coefficients for a single residue from the DDR memory and copy them in the BRAM. Eight lifting operations are computed by the two MAC-groups, i.e., four lifting operations per MAC-group, before writing back the result in the DDR memory.

In the case of $\mathsf{Lift}_{q\to Q}$, the result is a collection of 42×8 coefficients. This is because there are additional 42 moduli in Q and for each moduli there are eight coefficients. Hence 42 DDR-write operations are performed to copy the result in the DDR memory. Similarly after a $\mathsf{Lift}_{Q\to q}$ operation, the result is a collection of eight coefficients, each of size 1228 bits. Note that `WordDecomp` follows $\mathsf{Lift}_{Q\to q}$ in YASHE.Mult. So we write back the coefficients in the DDR

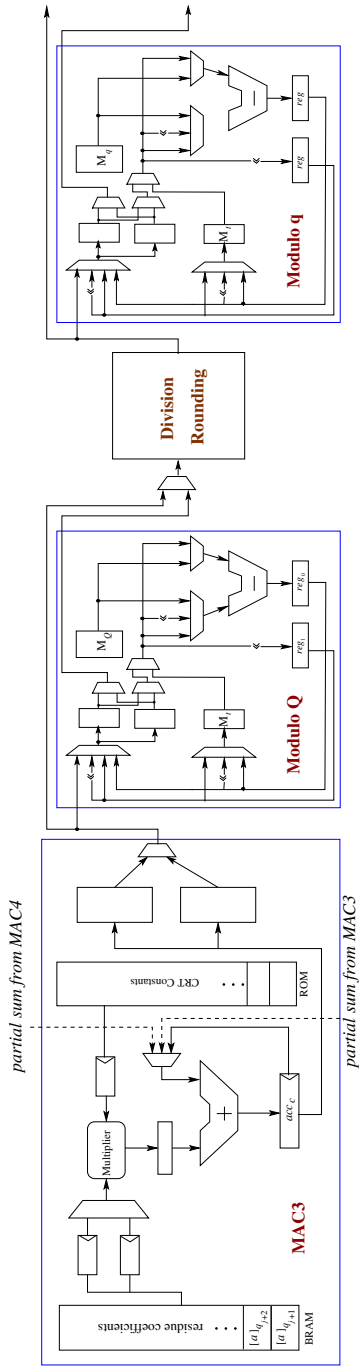

Figure 6.6: Unified architecture for $\mathsf{Lift}_{q\to Q}$ and $\mathsf{Lift}_{Q\to q}$.

Table 6.1: Area results on Xilinx Virtex-6 XC6VLX240T-1FF1156 FPGA

Resource	**Used**	**Avail.**	**Percentage**
Slice Registers	59,669	301,440	19.8 %
Slice LUTs	67,861	150,720	45.0 %
BlockRAM	80 BRAM36, 22 BRAM18	416	21.8 %
DSP48	232	768	30.2 %

Table 6.2: Latencies of the building blocks without DDR access overhead

Operation	**Cycles**
N-point NTT	47,795
N-point INTT	51,909
N-point-wise add/sub/mult	4,096
$\mathsf{Lift}_{Q\to q}$ (per coeff)†	687
$\mathsf{Lift}_{q\to Q}$ (per coeff)†	401
Poly mult in R_{q_j}	361,376

† Assuming pipeline processing of many coefficients

memory as words of length 60 bits. A total of 44 DDR-write operations are performed.

6.5 Results

We compiled the processor for the ML605 board which has a Virtex-6 FPGA XC6VLX240T-1FF1156. Different clock domains are used in the design: communication with the DDR memory is performed at 200 MHz, whereas computations are performed using a 100 MHz clock. The HE-coprocessor has $v = 16$ parallel cores for performing polynomial arithmetic, and two cores for computing the lifting operations. The area counts of our HE-coprocessor, including the DDR interface, are shown in Table 6.1.

Table 6.2 gives the latencies of the building blocks excluding the cost of DDR memory access. NTT and INTT computations are performed on polynomials of $N = 2^{16}$ coefficients. To save memory requirement, we compute the twiddle factors on the fly at the cost of N integer multiplications. One NTT computation using $v = 16$ cores requires $(N + \frac{N}{2}\log_2(N))/16 = 36,864$ multiplications.

However the computation of the twiddle factors in the pipelined data path of the PAU (Fig. 6.1) has data dependencies and thus causes bubbles in the pipeline. Following the design methodology of the public key encryption processor of Chap. 5, we use a small register-file that stores four consecutive twiddle factors, and reduce the cycles spent in the pipeline bubbles to around 10,000. In the case of an INTT, the additional cycles are spent during scaling operation by N^{-1}. To compute N-point-wise addition/subtraction/multiplication we need slightly more than 4,096 cycles.

Computation cost of the lifting operations: The cycle requirement for $\mathsf{Lift}_{Q\to q}$ is determined by the division-and-rounding operation, since it is the costliest computation in the pipeline of Fig. 6.6. If we assume that many $\mathsf{Lift}_{Q\to q}$ operations are performed in pipeline, then the cycle requirement per coefficient will be 687. However, due to the restrictions put by the DDR interface, we process only four $\mathsf{Lift}_{Q\to q}$ in pipeline (see Sect. 6.4.2). As a consequence 4,744 cycles are needed to process four coefficients. Similarly, when we assume that many $\mathsf{Lift}_{q\to Q}$ operations are performed in pipeline, cycle requirement per coefficient is 401. In practice, we can compute only four $\mathsf{Lift}_{q\to Q}$ in pipeline, and thus it takes total 2,016 cycles for computing four $\mathsf{Lift}_{q\to Q}$ operations.

Computation cost of polynomial multiplication modulo q_j: To multiply two residue polynomials modulo q_j, we compute two NTTs, then N-point-wise multiplications, and one INTT. The reduction of the result modulo $f(x)$ follows the Newton iteration method of Sect. 2.5.2. In this step, two NTTs, two N-point-wise multiplications, one $N/2$-point-wise subtraction and two INTTs are computed. Hence the computation of a polynomial multiplication in R_{q_j} requires four NTTs, three N-point-wise multiplications, one $N/2$-point-wise subtraction and three INTTs. This translates into 361,376 cycles.

Computation cost of YASHE.Mult: The cycle counts for the steps (see Sect. 6.3) are enumerated below.

1. To lift c_1 and c_2 (each having $N/2$ coefficients) from R_q to R_Q, we compute N $\mathsf{Lift}_{q\to Q}$ operations. This takes total 16,515,072 cycles using the two lifting cores, as each taking 2016 cycles to process four coefficients.

2. $\mathsf{PolyMultiply}_Q$ performs 83 polynomial multiplications in R_{q_j}. This translates into 29,994,208 cycles.

3. To bring the result back to R_q, $\mathsf{Lift}_{Q\to q}$ is performed on the $N/2$ coefficients. Since each lifting core takes 4,744 cycles to process four coefficients, it takes 19,431,424 cycles to lift the result back to R_q.

4. `WordDecomp` does not have computation cost since it only splits the large coefficients into words. In our architecture, this splitting is done

Table 6.3: Latencies and timings at 100/200 MHz computation/DDR clock

Operation	Computation cycles	DDR cycles	Total time
YASHE.Add	83,968	9,740,288	0.050s
YASHE.Mult	127,050,548	1,353,555,968	8.038s

automatically when we copy the data in the DDR memory from the FPGA.

5. In YASHE.KeySwitch summation of 22 polynomial multiplications is computed in R_q. Note that **evk** is constant, and hence can be kept in the NTT domain. For a single moduli, it requires 22 NTTs, 22 N-point-wise multiplications, 21 N-point-wise additions to get the summation of unreduced polynomial multiplications in the NTT domain. Now only one INTT is needed to get the unreduced result. For the reduction, two NTTs, two N-point-wise multiplications, one $N/2$-point-wise subtraction and two INTTs are computed. Hence, in total 24 NTTs, 24 N-point-wise multiplications, 21 N-point-wise additions, one $N/2$-point-wise subtraction, and 3 INTTs are performed for a single moduli. This translates into 61,109,844 cycles for all the 41 moduli.

Overall 127,050,548 cycles are spent in YASHE.Mult. At 100 MHz clock frequency, this corresponds to 1.27 seconds.

Overhead of the DDR memory access: The DDR interface reads or writes 256 bits in a single burst. For YASHE.Mult, the communication with the DDR memory takes around 1,353,555,968 cycles at 200 MHz. For YASHE.Add, the number of cycles spent in the DDR access is around 9,740,288.

Table 6.3 shows the timing requirement for computing YASHE.Add and YASHE.Mult operations including the overhead of DDR memory access. Based on the timing of YASHE.Mult we see that the designed architecture would take roughly 11316s, which is about 3h and 9m to evaluate SIMON-64/128 (44 rounds with 32 ANDs). Since the chosen parameter set processes 2048 slots in SIMD, the per-block timing will be roughly 5.5s.

Discussions: Lepoint and Naehrig [77] presented C++ implementations for homomorphic evaluations of SIMON 64/128 with YASHE running on a 4-core Intel Core i7-2600 at 3.4 GHz. They reported computation times of 4193s for SIMON-64/128 using all 4 cores. With respect to their software implementation, our hardware implementation is 2.7 times slower. In this work our focus was on designing the computation core of the YASHE; the DDR memory interface

is a proof of concept implementation. With 256-bit burst data width, the DDR interface offers a only 1.97Gb/s read speed and hence becomes the main bottleneck in our implementation. Desktop computers have industry-optimized DDR interface, and the Intel Core i7-2600 processor has 8MB cache memory [60]. Since a ciphertext is of size 4.8MB, that the overhead of memory access in [77] would be much lower than ours.

We have implemented a new DDR memory interface with 2048-bit burst data length. The interface provides 10Gb/s read and 27Gb/s write speed, and hence with this interface, the memory access overhead would become almost equal to the computation cost. This would allow us to reduce the overall time by performing the memory access in parallel with computation using two sets of BRAMs in the FPGA: when one set is used for the computation, the other set is used for the memory access. We consider the integration of the new memory interface in the HE-coprocessor as a future work.

The ML605 board has a medium size FPGA. To know the amount of speedup that we can achieve using large FPGAs, we have designed a new architecture that keeps eight instantiates of the computation core of the present architecture. The new architecture brings down the *computation cost* roughly by a factor of eight. On a Xilinx Virtex-7 FPGA XC7V1140T, which is the largest device of the Virtex-7 FPGA family, the parallel architecture consumes 23 % of registers, 53 % of LUTs, 53 % of DSP slices, and 38 % of BlockRAM memory. It will be interesting to know the effect of DDR memory access in the overall computation time.

6.6 Summary

In this work we designed the hardware building blocks for homomorphic evaluation of small-complexity algorithms using the YASHE scheme. We showed that FPGAs can accelerate the computation intensive operations during a homomorphic function evaluation. Despite this, we found that a massive amount of data exchange takes place between the FPGA and the external DDR memory during a homomorphic function evaluation. This is because only a part of the ciphertext can be fit in the internal memory of the FPGA. The interface with the DDR memory plays a very important role in the performance and may even become a bottleneck unless it is implemented with special care.

We presented a single-FPGA design of homomorphic evaluation with YASHE. An obvious way to improve the performance would be to use a multi-FPGA design (a cluster). We see several parallelization approaches. The first and simplest option is to instantiate parallel FPGAs so that each of them computes

different homomorphic evaluations independently of each other. This approach improves throughput, but the latency of an individual evaluation remains the same. The second option is to distribute the residue polynomial arithmetic into several FPGAs since they can be computed independently. This approach will reduce the latency. However, the lifting operations require the coefficients from different residue polynomials during the computation of the sum of products. To do this, inter-FPGA communication will be needed. The third option is to divide different parts of a homomorphic multiplication to different FPGAs and perform them in a pipelined fashion in order to increase throughput. The fourth option is to mix the other three options and it may lead to good tradeoffs that avoid the disadvantages of the other options. The techniques represented in this research can be extrapolated to support these options.

Other relevant works.

Concurrent to our initial version of the work, Pöppelmann et al. [107] and Doröz et al. [36] designed hardware accelerators for SHE. For a parameter set with $n = 16384$ and a 512-bit prime q, capable of evaluating 9 levels of YASHE.Mult, the hardware accelerator by Pöppelmann et al. [107] takes 48.67ms to compute a homomorphic multiplication on a data-center accelerator infrastructure called Catapult [108], which has a medium size Stratix V GS D5 (5GSMD5) FPGA. Their parameter set is smaller in size than ours and allows the negative-wrapped convolution for polynomial multiplication, but lacks the SIMD feature. Doröz et al. [36] applies CRT similar to us and uses a large Virtex-7 FPGA to compute the polynomial multiplications only. They use a high-speed PCIe interface to compute the other operations on a desktop computer. They evaluate the AES block cipher and estimate a per block of 442 msec using the LTV [83] scheme.

Recently in 2016, Albrecht et al. [6] published a subfield lattice attack that runs in sub-exponential time on overstretched NTRU assumptions. Since the key generation part of the YASHE scheme relies on a mildly overstretched NTRU assumption, the subfield attack makes YASHE insecure. We would like to remark that this attack does not make our entire work invalid since the hardware architecture can reused to evaluate other ring-LWE based homomorphic encryption schemes. For example, the FV scheme (see Sect. 2.4.2), which is secure against known cryptanalysis methods, can be evaluated using our architecture after a small modification. The modification is required to support the `FV.Relin` operation, as c_0 and c_1 require computation of reqidues modulo q_i from coefficients modulo q. At the moment our HE-coprocessor does not support this operation.

Chapter 7

Recryption-box assisted homomorphic function evaluation

CONTENT SOURCES:

Sujoy Sinha Roy, Frederik Vercauteren, Jo Vliegen, and Ingrid Verbauwhede Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search Accepted in *IEEE Transactions on Computers*. Preprint available on IEEE Xplore, DOI: 10.1109/TC.2017.2686385.

Contribution: Main author.

7.1 Introduction

A somewhat homomorphic encryption (SHE) scheme can be used to compute on the encrypted data, but each operation increases the noise inherent in the ciphertexts. Once the noise reaches a certain threshold that depends on the parameters of the scheme, decryption will fail. Gentry's FHE scheme uses an SHE scheme combined with a mechanism known as *bootstrapping*. The bootstrapping operation is used to publicly "refresh" a noisy ciphertext and repeated application enables evaluations of arbitrary depth. However FHE is very slow even after orders of magnitude improvement in [21, 22, 30, 137, 43, 47,

49, 91]. The main problem is that the bootstrapping operation is only possible if the parameters of the SHE scheme are chosen large enough to accommodate for the bootstrapping operation. This requirement makes the bootstrapping operation tremendously slow, e.g. for [25] it takes 172 seconds on an Intel Core-i7 processor. Hence it is not yet possible to deploy FHE in cloud computations. Even SHE schemes that can evaluate functions of small complexity take a large amount time. For e.g., evaluation of one SIMON-64/128 decryption on encrypted data takes more than an hour on a 4-core Intel Core-i7 processor [76]. Moreover from the performance results presented in the last chapter we find that even with a powerful hardware accelerator, evaluation remains too slow to use in practical applications.

In this chapter we propose a scheme to perform homomorphic evaluations of arbitrary depth in much less time with the assistance of a special module *recryption-box*. Our solution is to bypass the costly bootstrapping operation using a third party recryption-box that is instantiated in two different setups. In the first setup the recryption-box uses a key switching technique that allows the cloud server to convert a ciphertext encrypted under user's public key into a ciphertext encrypted under box's public key. With this, the box performs a decryption using its own private key and then a re-encryption using user's public key. Naturally the large noise in the encrypted data is eliminated. In the second setup a multiparty computation scheme is used: noisy ciphertexts are decrypted among multiple parties, and then reencrypted again. This re-encryption operation gives a freshly encrypted data with limited noise as the shared multiparty decryption operation removes the large noise inherent in the ciphertexts. During the execution of an application on encrypted data, the cloud performs the homomorphic operations, and then sends the dirty ciphertexts to the third party recryption-box (or boxes).

We implement the recryption-box on a Xilinx Virtex 6 FPGA board ML605. The recryption-box is connected to the cloud computer over the internet using the Ethernet interface. During a recryption operation, the cloud computer sends noisy (masked) encrypted data to the recryption-box, which then returns refreshed encrypted data. To know the effect of the recryption-box model on the run time, we have implemented *encrypted search* as the target application. In an encrypted search, clients send encrypted queries to the search engine, and the search engine returns encrypted results. Neither the search engine, nor the other parties come to know about the client's search queries. We show that with the usage of the recryption-boxes we could reduce the encrypted search time by an order of magnitude. Although we instantiate the recryption-box on an FPGA, we note it is possible to implement it in a trusted execution environment or using specialized instructions such as SGX, assuming sufficient countermeasures are taken against physical attacks.

The chapter is organized as follows. Sect. 7.2 describes the recryption-box and its instantiations. The recryption-box is used in Sect. 7.3 to assist an encrypted search algorithm. Sect. 7.4 gives implementation details of the recryption-box and describes the optimization techniques. Experimental results are provided in Sect. 7.5. The final section draws conclusions.

7.2 Instantiations of the recryption-box

In this section, we describe two possible instantiations of the recryption-box and analyze their security and ease of use. In all cases, the parameters of the scheme become independent of the minimum size required for bootstrapping, resulting in faster homomorphic evaluations overall. We also consider possible applications of the setup described in this chapter.

Most homomorphic encryption schemes (including FV) admit an operation called key switching. Key switching allows to transform a ciphertext encrypted under one public key, into a valid ciphertext encrypted under a different public key. More in detail: assume the recryption-box has its own private/public key pair (s_r, b_r, a_r) and the i-th user's keypair is (s_i, b_i, a_i). The user can then compute the key switching key as follows: he samples a vector $\mathbf{u}$ of l elements uniformly from χ_{key} (in our case a signed binary distribution), and two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ of l elements from χ_{err}. Next he computes the key switching key $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\} = \{\texttt{PowersOf}_{w,q}(s_i) + \mathbf{u} \cdot b_r + \mathbf{e}_1 \in R_q^l, \mathbf{u} \cdot a_r + \mathbf{e}_2 \in R_q^l\}$. The $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\}$ together with $\{b_i, a_i\}$ is sent to the cloud. The cloud uses the key switching key to switch a ciphertext $\{c_{0_i}, c_{1_i}\}$ encrypted under the user's public key to a valid ciphertext $\{c_{0_r}, c_{1_r}\}$ encrypted under the box's public key as follows: $\{c_{0_r}, c_{1_r}\} = \{\langle \texttt{WordDecomp}_{w,q}(c_{1_i}), \mathbf{ksk}_{0_i} \rangle + c_0, \langle \texttt{WordDecomp}_{w,q}(c_{1_i}), \mathbf{ksk}_{1_i} \rangle\}$. Before sending the ciphertext $\{c_{0_r}, c_{1_r}\}$ to the box, the cloud additively masks it (using the fact that the scheme is additively homomorphic) to obtain $\{c'_{0_r}, c'_{1_r}\}$. The ciphertext $\{c'_{0_r}, c'_{1_r}\}$ together with user's public key $\{b_i, a_i\}$ is sent to the box, who decrypts it using its own private key and freshly encrypts it using the user's public key. The resulting ciphertext $\tilde{c}_i$ is then sent back to the cloud, who removes the additive mask it added before.

For the above setup to offer any security at all, the following assumptions have to be made: firstly, we assume that both the box and cloud are honest but curious. In particular, the cloud has to apply a random mask before sending the ciphertext to the box such that it cannot recover the underlying plaintext. And in turn, the box has to execute the encryption correctly by choosing random error polynomials. Secondly, we assume that the cloud and box do not collude, e.g. the key switching key $\{\mathbf{ksk}_{0_i}, \mathbf{ksk}_{1_i}\}$ should not be given to the box since

it would allow the box to derive the private key of the user. The advantage of this setup clearly is that a single recryption-box can deal with many users. The downsides are the slightly stronger security assumptions and the extra operations involved such as key switching and additive masking by the cloud.

The second instantiation does not rely on a key switching key and makes it much more difficult for the cloud and the recryption-box to collude by using a threshold scheme to split the secret key over several parties and using a distributed decryption protocol. The idea has similarity with the work in [26] where the authors interpolate between multiparty computation and fully homomorphic encryption. The secret key can be split using a th out of n Shamir threshold sharing over the ring R_q. The i-th party receives a share $s_{r_i} \in R_q$ that equals the evaluation of a random polynomial $p(x)$ of degree $th-1$ in a public value $a_i \in R_q$ assigned to each party (one could sometimes even take $a_i = i$), i.e. $s_{r_i} = p(a_i)$. The secret key s of the user can then be obtained as $s = p(0)$, and it is clear that any set of th valid shares allows to recover s using for instance Lagrange interpolation. Denote the i-th Lagrange multiplier (for the set of th contributors) by $\lambda_i = \prod_{j \neq i} a_j/(a_j - a_i)$, then s can be recovered as $s = \sum_i \lambda_i s_{r_i}$. Denote by $\tilde{s}_{r_i}$ the scaled share $\lambda_i s_{r_i}$, then s can be simply recovered as $s = \sum_i \tilde{s}_{r_i}$. The main advantage of using Shamir secret sharing is that it defines a ring homorphism between $R_q, +, \cdot$ and $R_q^{th}, +, \cdot$. In particular, any algebraic expression in R_q can be recovered from executing the same expression on each of the th shares and reconstructing the result using interpolation.

The distributed decryption protocol will work in two steps. In the first step, for a given ciphertext (c_0, c_1) each party computes $d_i = \tilde{s}_{r_i} \cdot c_1 + e_i$ where e_i is a Gaussian distributed error polynomial. Note that recovering $\tilde{s}_{r_i}$ from d_i is hard since this corresponds precisely to the ring-LWE problem, so one party cannot recover another party's share. The shares d_i are then distributed over an encrypted channel to the other parties. In the second step, each party adds the shares to recover $c_1 \cdot s + e$. Now the end-party (or any party) then recovers the message m as $m = \lfloor \frac{t}{q} \cdot [c_0 + s \cdot c_1 + e]_q \rceil \in R_t$ and returns a fresh encryption of m as the final result. Since the end-party recovers the message m, it is required that the server additively masks the message before sending it to the recryption-boxes. The security of the system now relies on the fact that not more than $th-1$ parties collude with the cloud server and that the cloud server uses additive masking.

7.3 Encrypted search

To know the effect of the proposed recryption-box in real life, we have implemented encrypted search as the target application. The reason behind this particular choice is mainly because of its future prospects. In an encrypted search, a client sends an encrypted keyword to the search engine and then the search engine returns the search-response in an encrypted format to the client. Due to its *encrypted* nature, the search engine remains oblivious of the search keyword. Such an encrypted search application is possible when the encryption scheme is homomorphic.

In the search engine, the search results are stored in unencrypted format in a table (as table entries) indexed by the numeric representations of the search keywords. During an encrypted search, a client's encrypted keyword is compared with the encryptions of the table indexes, and then the comparison results are used to perform arithmetic on the encryptions of the table entries. Since the search table is in unencrypted format, the search engine needs to encrypt the entire table under the public key of the client in order to perform homomorphic operations. However in reality the search engine needs to encrypt only two bits instead of the entire table. The client sends her public key along with the encrypted keyword to the search engine. Next, the search engine encrypts bit-0 and bit-1 using the public key of the client and constructs encryptions of the search table indexes and the entries by simply replacing the plaintext bits with the encryptions of bit-0 and bit-1. After the completion of the search operation, the result is an encryption of the search result that is associated with the search keyword. The total amount of data exchange between the client and the search engine is: the public key of the user, the homomorphic encryption of the search keyword, and the homomorphic encryption of the search result.

There are several algorithms to search in a plaintext database. The most efficient ones such as binary search or half-interval search [146] have logarithmic complexity. But in the case of an encrypted search, the search algorithm has linear complexity as it does not see the search term in plaintext and thus has to go through all of the database entries. A linear encrypted search operation is shown in Algorithm 16. The table to be searched is represented as an array of 2^l elements and the elements are accessed using the l-bit *index* variable. We assume that the table entries (i.e. the search data in any location) are of p bit. The encrypted search keyword K is a string of l ciphertexts k_i. In the start phase, the algorithm computes the encryptions e_0 and e_1 of bit-0 and bit-1 respectively (line 2 and 3), and then initializes the accumulator R to a string of p encryptions of bit-0 (line 4). Next, in the search phase the *for*-loop goes through all the indexes of the table. In line 6 the algorithm constructs an encryption of the one's complement of *index* (i.e. $\overline{index}$) in the variable C by

Algorithm 16: *Linear encrypted search*

Input: Encrypted search keyword $K = \{k_{l-1} \dots k_0\}$ and the client's public key pk
Output: Encrypted search result $R = \{r_{p-1} \dots r_0\}$

```
1  begin
2      e_0 ← enc(0, pk);
3      e_1 ← enc(1, pk);
4      R ← {e_0, ..., e_0};
5      for index = 0 to 2^l − 1 do
6          C ← {e_~index_{l−1}, ..., e_~index_0} ;        /* enc. of ~index */
7          T ← {add(k_{l−1}, c_{l−1}), ..., add(k_0, c_0)};
8          b ← t_0;
9          for i = 1 to l − 1 do
10             b ← mul(b, t_i);
11         end
12         D ← table[index];
13         for i = 0 to p − 1 do
14             if d_i = 1 then
15                 r_i ← add(r_i, b) ;                    /* i-th bit of R */
16             end
17         end
18     end
19 end
```

concatenating the encryptions of the bits of $\overline{index}$. Next the algorithm adds the encrypted one's complement of the index with the encrypted keyword and obtains T, and then cumulatively multiplies the l encrypted bits of T to get a single encrypted bit b (lines 8-10). Note that b will be an encryption of bit-1 only for the loop $index$ that is equal to the unencrypted search keyword; otherwise b will be an encryption of bit-0. Now the algorithm fetches the table entry and stores it in the variable D. In the next part of the loop, the algorithm adds b for each nonzero bits d_i of D with the encrypted bits r_i of the accumulator R. Note that only when the unencrypted search keyword matches with the index of the search table, this b is an encryption of bit-1, and hence an encryption of the associated search result is added with the R; for all other indexes, encryptions of zeros are added with the accumulator. In this way when the for-loop covers all the indexes of the search table, we get an encryption of the search result in R.

Faster linear search on encrypted data

The most costly part in Algorithm 16 is the homomorphic multiplication of the l encrypted bits of T in lines 8 to 10. We reduce the number of homomorphic multiplications with the help of a window based pre-computation technique shown in Algorithm 17. With a window size w, the given encrypted keyword is split into l/w chunks. For simplicity let us assume that l is a multiple of w. Then for each chunk of the encrypted keyword, a pre-computation table is

Algorithm 17: *Precomputation in linear search*

Input: w-bit chunk of encrypted keyword $K = \{k_{w-1} \dots k_0\}$ and client's public key pk
Output: Precomputation table $precomp[]$ with 2^w entries

```
1  begin
2      e_0 ← enc(0, pk);
3      e_1 ← enc(1, pk);
4      for index = 0 to 2^w − 1 do
5          C ← {e_~index_{w−1}, ..., e_~index_0};
6          T ← {add(k_{w−1}, c_{w−1}), ..., add(k_0, c_0)};
7          b ← t_0;
8          for i = 1 to w − 1 do
9              b ← mul(b, t_i);
10         end
11         precomp[index] ← b;
12     end
13 end
```

constructed. The algorithm considers all 2^w w-bit indexes: first an encryption of the one's complement of the index is added to the keyword-chunk to obtain T, and then the w encrypted bits in T are multiplied together to obtain the ciphertext b. Next b is stored in the precomputation table.

The search algorithm 18 uses the pre-computation tables to speedup computation. The loop *index* is split into l/w chunks in lines 4-5, and then the pre-computation tables corresponding to the chunks are accessed in line 6 to perform the cumulative multiplications in line 9. In comparison to Algorithm 16 this algorithm reduces the number of homomorphic multiplications within the search loop by a factor w. The new algorithm requires additional memory to store the $2^w \frac{l}{w}$ precomputed table entries.

Comparison with private information retrieval (PIR)

A PIR protocol allows a user to retrieve an item from a server without revealing any information about the item. An l-server PIR protocol, such as the protocol by Goldberg et al. [50], will be faster than our encrypted search algorithm. But our encrypted search has several advantages as follows. Our protocol is less interactive: the user sends an encrypted query to the search engine and then waits for the encrypted search result. In the l-server PIR protocol the user sends queries to all the servers. The amount of data exchange increases with the number of servers. Whereas in the encrypted search this is determined by the ciphertext expansion ratio of the homomorphic encryption scheme. Thus our encrypted search has an asymptotically lower communication cost. Note that a lot of communication happens between the search engine and the recryption-boxes during the pre-computation phase of an encrypted search operation. It

Algorithm 18: *Precomputation assisted linear encrypted search*

Input: Encrypted search keyword $K = \{k_{l-1} \dots k_0\}$ and the client's public key pk
Output: Encrypted search result $R = \{r_{p-1} \dots r_0\}$

```
1  begin
2      R ← {e_0, ..., e_0};
3      for index = 0 to 2^l − 1 do
4          for i = 0 to l/w − 1 do
5              chunk_i ← {index_{iw+w−1} ... index_{iw}} ;
6          end
7          b ← precomp_0[chunk_0];
8          for i = 1 to l/w − 1 do
9              temp ← precomp_i[chunk_i];
10             b ← mul(b, temp);
11         end
12         D ← table[index];
13         for i = 0 to p − 1 do
14             if d_i = 1 then
15                 r_i ← add(r_i, b);
16             end
17         end
18     end
19 end
```

is quite feasible to bring this cost down by keeping the recryption-boxes in a dedicated very high-speed network (e.g. Terabit Ethernet). In the l-server PIR protocol we cannot assume that all the users are connected to the servers by a dedicated high-speed network. One problem with the l-server PIR protocol is that the protocol assumes that the servers possess copies of the database. In practice due to several business-related issues the internet search companies cannot share their databases with the server parties. In our approach this is not a problem as the recryption-boxes are used only to perform computation on masked encrypted data. The other advantage is that with a little effort we can turn our encrypted search algorithm into an encrypted database update algorithm. With PIR this is not possible.

There are many single database PIR protocols in the literature. But their implementations are slow. In [87] Melchor et al. presents performance results of three PIR protocols for a database of 1000 elements, each of size 2MB. On a Core 2 Duo processor their implementations of the PIR protocols from [79] and [48] take around 33 hours and 17 hours respectively. Their own lattice-based PIR protocol takes 10 minutes. In [150] Yi et al. proposes a practical PIR based on a variant of the DGHV somewhat homomorphic encryption scheme [138]. Their implementation of the PIR for a 60-bit security parameter set reduces the computation cost by an order of magnitude and takes only one minute for the 1000 element database. Our encrypted search is faster than the PIR protocol of Yi et al.

7.4 Implementation

In this chapter we implement a fast architecture for the proposed recryption-box and then use this box to assist an encrypted search engine running on a server machine. The search algorithm is written in high level C and the recryption-box is implemented as a hardware module running on Xilinx ML605 board. The search engine performs homomorphic evaluations on the encrypted data. When the number of homomorphic evaluations reaches the maximum depth supported by the parameter set of the homomorphic encryption scheme, the search engine blinds the encrypted data and then sends it to the recryption-box over a Gigabit Ethernet channel. After a recryption, fresh encrypted data is sent back to the search engine.

7.4.1 Parameter set used in the implementation

We use the FV scheme [43] that we described in Sect. 2.4.2 as the homomorphic encryption scheme. We target fastest computation time and at least 90-bit security. Since the computation time of the FV homomorphic encryption scheme has almost a quadratic-complexity with respect to the multiplicative depth, we chose a parameter set that supports the minimum multiplicative depth, i.e. the depth one. Following [76] we set the dimension of the polynomial ring R_q to $n = 1024$, the modulus q to a 40-bit integer, and the parameter s of the discrete Gaussian distribution to 11.32. This parameter set has 96-bit security [7]. Similar to the last chapter we work in the polynomial ring $R = \mathbb{Z}[x]/(f(x))$, but with the irreducible polynomial $f(x) = x^n + 1$. The irreducible polynomial does not allow SIMD, and hence only one bit is encrypted in a ciphertext.

7.4.2 Algorithmic optimizations for efficient architecture

The basic computations in the FV encryption and decryption (Sect. 2.4.2) are discrete Gaussian sampling, polynomial addition and multiplication, and decoding-encoding. Among the arithmetic operations, polynomial multiplication is the costliest one. We use the memory efficient NTT Alg. 11 from Sect. 5.3.3 to perform polynomial multiplication in the most efficient way. For the chosen parameter set, integer arithmetic operations are performed with respect to a 40-bit modulus q. To achieve faster processing through parallelization, we use the *Chinese Remainder Theorem* (CRT) to split 40-bit arithmetic into two parallel 20-bit arithmetic operations. We take the modulus q as a product of two 20-bit primes $q_0 = 878593$ and $q_1 = 890881$. Since both q_0 and q_1 are congruent to 1 modulo $2n$, we use the negative-wrapped convolution for faster

NTT computation. With the application of CRT each operation modulo q during a polynomial arithmetic turns into two parallel 20-bit operations modulo q_0 and q_1. Note that the security of the encryption scheme does not get affected by this choice for q.

This parallel nature of the algorithm is very useful since the underlying hardware platform is also parallel. Hence two computation threads modulo q_0 and q_1 run in parallel. Beside this parallel processing, there is an another advantage of splitting the computation into two half-sized integers. Xilinx Virtex 6 FPGAs have fast but small 25×18 DSP multipliers. Hence a 20-bit coefficient can be easily processed by the small DSP multipliers (with some additional logic elements). Moreover smaller integer size is also very helpful to keep a pair of coefficients of a residue polynomial in one BRAM address and reduce the memory access overhead by using Alg. 11. We need only one 36K BRAM slice to store a residue polynomial with two coefficients in one address.

Though CRT allows parallel processing, it has the overhead of inverse-CRT computation whenever the computation demands arithmetic in modulo q. For the proposed recryption-box, inverse-CRT is required only during the decoding phase of the FV decryption operation. This is because the decoding operation needs to compare the coefficients with $q/4$ and $3q/4$. However, for our application this inverse-CRT computation is actually not a major overhead since we only need to decode the least significant coefficient of the ciphertext polynomial, and it is known that the remaining coefficients decode to zeros. This is because of the fact that we encrypt only one bit in a ciphertext using the FV homomorphic scheme, and the encrypted bit remains in the least significant coefficient of the ciphertext. Hence we compute the inverse CRT only once. Note that all of the remaining arithmetic operations in the FV encryption and decryption can be performed on the CRT-represented shares.

7.4.3 Architecture

The internal architecture of the processor part of the recryption-box is shown in Fig. 7.1. The processor has two symmetric polynomial arithmetic and logic units (PALUs) for performing residue arithmetic modulo q_0 and q_1 in parallel. The PALUs are designed following the footprint of the compact ring-LWE encryption architecture in Sect. 5. Each PALU is connected to a memory file that keeps the residue polynomials.

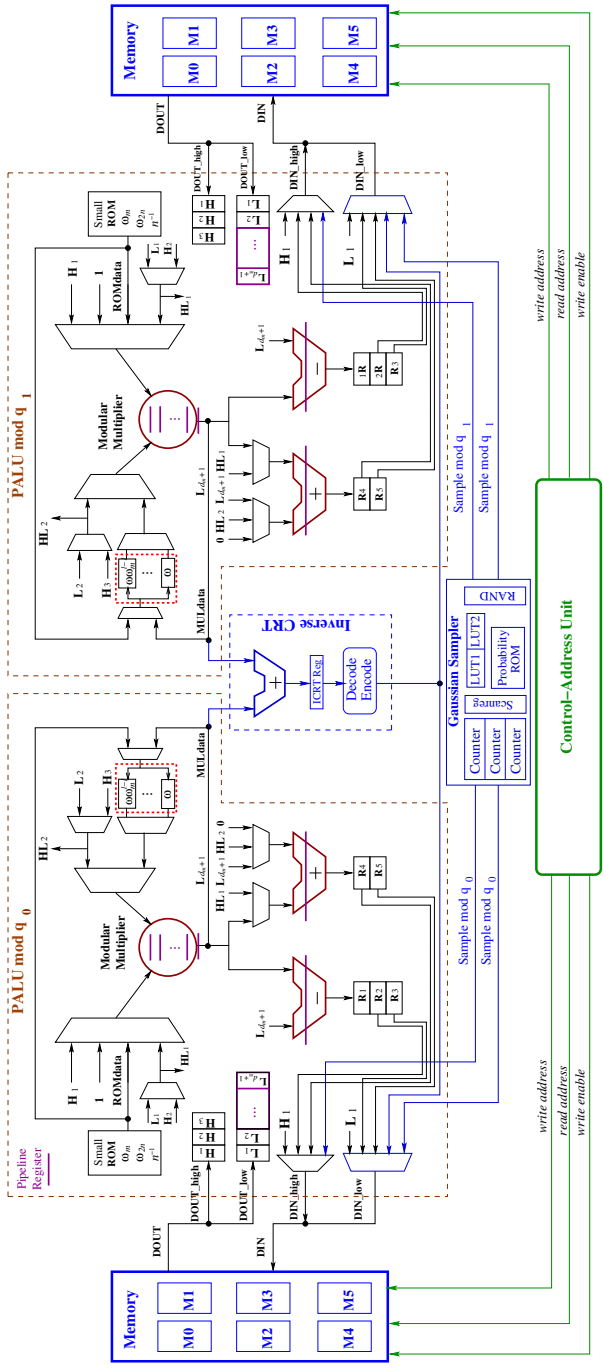

Figure 7.1: Architecture of The Recryption-box

PALUs

Each PALU has modular multiplier, modular addition and subtraction circuits to perform arithmetic operations on the input coefficients during a polynomial operation. The integer multipliers inside the modular multipliers are implemented using DSP multipliers. For the modular reduction of the integer multiplication result, we have used window based modular reduction technique. This modular reduction technique does not depend on the choice of the 20-bit prime modulus and is thus generic. The critical path of the PALU is through the modular multiplier and then through the addition (or subtraction) circuit. We split the critical path in almost equal delay sections using pipelines and achieve high operating frequency.

During an NTT computation on a residue polynomial, the control logic follows the *memory efficient NTT* Alg. 11 and processes two coefficient-pairs $(A[k+j+m/2], A[k+j])$ and $(A[k+m+j+m/2], A[k+m+j])$ (i.e. four coefficients) simultaneously excluding the last loop. In this architecture we do not store the fixed twiddle factors, and instead compute them on the fly during an NTT operation. The small ω-ROM stores the $\log(n)$ twiddle factors ω_m to generate the new twiddle factors. The modular multiplier circuit is reused for computing coefficient wise multiplications, and the modular addition/subtraction circuit is reused for coefficient wise addition/subtraction operations during polynomial arithmetic.

7.4.4 Inverse CRT

The inverse CRT combines two residues and computes the coefficient modulo q. Let a_0 and a_1 be the two residues. Then the equation for the inverse CRT computation is shown below.

$$a = [a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1 + [a_1 \cdot q_0^{-1}]_{q_1} \cdot q_0 \mod q \ . \tag{7.1}$$

In the above computation $[q_1^{-1}]_{q_0}$, q_1, $[q_0^{-1}]_{q_1}$, and q_0 are constants. Hence we store these constants in the PALUs. The PALU for the residue q_0 computes $[a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1$ part of Eq. 7.1 and the other PALU computes the remaining part. The first PALU computes in two steps: first it computes $[a_0 \cdot q_1^{-1}]_{q_0}$ using the modular multiplier, then it multiplies this 20-bit result with q_1 to get the 40-bit integer multiplication result $[a_0 \cdot q_1^{-1}]_{q_0} \cdot q_1$. The other PALU associated with the modulus q_1 computes $[a_1 \cdot q_0^{-1}]_{q_1} \cdot q_0$ in a similar way. These two 40-bit outputs from the two PALUs are added together using the adder in the *Inverse CRT* block (Fig. 7.1). Next the 41-bit addition result is reduced by the 40-bit q by performing one subtraction. The *Decode-Encode* block in Fig. 7.1 compares

the inverse CRT output with $q/4$ and $3q/4$ and then encodes the coefficient to either 0 or $q/2$. Note, that this inverse CRT is performed only once as we need to decode-encode only the least significant coefficient in the FV decryption.

7.4.5 The memory

The memory of the recryption-box consists of two independent memory files (Fig. 7.1) for the two PALUs. Each of the two memory files contains six RAM blocks M0, M1, M2, M3, M4 and M5, each containing 512 words. The coefficients of residue polynomials are kept as pairs in these RAM words. Each of these six RAM blocks consumes one 36K BRAM slice. During a recryption operation, the box's share of the client's secret key is loaded in M0, and the clients public key is loaded in M1 and M2. Since these keys are constants for a client, they are kept in the NTT domain to avoid unnecessary computation.

In the decryption phase, RAM blocks M3 and M4 are used to store the two polynomials c_0 and c_1 of the ciphertext. Hence a forward NTT of M3 is computed followed by a coefficient-wise multiplication of M0 and M3, and then the result is stored in M3. Next an inverse NTT is computed on $M3$ and this ends the polynomial multiplication in the multiparty decryption operation.

During the encryption phase, the encoded message is kept in M3. The noise polynomial e_1 is generated in M4, then added with the encoded message and finally the result is kept in M3. Another noise polynomial u is generated in M5. The two polynomial multiplications $pk_0 \cdot u$ and $pk_1 \cdot u$ are kept in M4 and M5.

7.4.6 The discrete Gaussian sampler

The sampler uses the Knuth-Yao random walk algorithm. It is borrowed from the `LPR` ring-LWE public key encryption processor of Chap. 5.

7.4.7 The ethernet communication unit

The Xilinx ML605 development board has a single physical networking interface which is wired to the FPGA. This FPGA has four Embedded Tri-Mode Ethernet MAC cores [148] which are present in the silicon of the FPGA (hard-cores). To incorporate such a core in the design, the Xilinx CORE Generator is used to provide wrapper files which help to configure and interface the Ethernet MAC. Because high throughput is required and all the wiring between the

physical interface (PHY) and the MAC on the FPGA are present, the Gigabit media-independent Inteface (GMII) is used.

Using the wrapper files (as generated by the Xilinx IPCore tool) to interface the hard-core MAC provides an easy-to-use interface, consisting of four signals: the data vector (8-bit wide), a data-valid signal, a data-user, and the data-last signal. For a more detailed explanation on these signals and their usage, the reader is referred to the Xilinx documentation [148, 147].

7.5 Results

We have implemented the encrypted search (Algorithm 18) as the target application for performance evaluation. The search algorithm is a software program written using high-level C with GMP library for long integer arithmetic, and runs on a powerful server that has Intel(R) Xeon(R) CPU E5-2687W v3 with 40 cores running at 3.10GHz. During an encrypted search operation, the search engine, i.e. the server machine sends noisy ciphertexts to the third party recryption-boxes over a gigabit Ethernet channel. The recryption-boxes are implemented on Xilinx FPGA boards ML605, which comes with a medium-size Virtex 6 FPGA XC6VLX240T and a gigabit Ethernet for the external communication [149].

In this implementation, we have restricted the size of the search table to 2^{16} entries. Thus the index of the search table has 16 bit width. We use an 8-bit window size in Algorithm 18. With this window size, the *index* variable in Algorithm 18 consists of two chunks, and hence only one homomorphic multiplication is required per iteration of the search loop. This is also very helpful in reducing the network traffic between the search engine and the recryption-boxes, because the parameter set of the homomorphic scheme supports only one multiplication, and with only one multiplication per iteration of the search loop, the search engine does not need to refresh its encrypted data. The recryption-boxes are required only during the precomputation phase of the encrypted search operation (Algorithm 17). Since this precomputation phase has a very small computation overhead with respect to the actual search loop, the network communication overhead is not a big issue.

The high level software implementation takes 21 seconds to perform one encrypted search operation. To know the actual effect of the proposed recryption-box assisted encrypted search model, we have also implemented an encrypted search software that does not use the recryption-boxes. The parameter set for this implementation supports the full multiplicative depth of the encrypted search, and has polynomial dimension $n = 4096$, modulus size 141 bits, and

Table 7.1: Area of the recryption-box on Xilinx Virtex-6 XC6VLX240T-1FF1156

Component	Resource	Used	Avail.	Percentage
recryption-box	Slice Registers	2,684	301,440	0.9 %
	Slice LUTs	3,379	150,720	2.3 %
	BlockRAM36k	12	416	2.9 %
	DSP48	4	768	0.5 %
Processor	Slice Registers	1,848	301,440	0.6 %
	Slice LUTs	2,751	150,720	1.8 %
	BlockRAM36k	12	416	2.9 %
	DSP48	4	768	0.5 %

Gaussian distribution parameter $s = 11.32$. The security analysis following [7] gives 96 bit security for this parameter set. The software takes 6 minutes and 40 seconds to perform an encrypted search on the same server machine, which is roughly 20 times slower than with the recryption-box.

We have used mixed Verilog and VHDL to describe the recryption-box architecture and have compiled the architecture using the Xilinx ISE 14.7 tool with a constraint file. The area requirements of the recryption-box architectures are shown in Table 7.1. The processor part of the recryption-box consumes around 1.8% of the slice LUTs and 0.6% of the registers available in the FPGA. For the recryption-box architectures, the additional area requirement is mostly due to the Ethernet wrapper and the small components that are used to perform the communication between the FPGA and the computer.

The latencies of different operations are shown in Table 7.5. In the design constraint file the operating frequencies of the clocks were set to 125MHz; both the Ethernet wrapper and the arithmetic unit run at 125 MHz, but using different clock domains. From the table we see that the most computation intensive operations are NTT and INTT. A decryption operation computes one NTT, one coefficient wise multiplication and one INTT, one coefficient wise addition, inverse CRT followed by a decode-encode of one coefficient. Thus it takes around 0.153 ms. An encryption computes additional discrete Gaussian sampling operations and thus takes slightly more amount of time that the decryption operation. One recryption operation is basically a decryption followed by an encryption and thus takes around 0.428 ms, excluding the cost of data transfer between the FPGA and the host computer. To know the overall time (data exchange + computation) of one recryption operation, we measured the actual timing from the FPGA board using a counter, and found this to be 0.6 ms.

Table 7.2: Latencies and timings at 125 MHz

Operation	Clocks	Time
NTT	7181	0.0575 ms
INTT	9910	0.0793 ms
Coefficient wise add	1032	0.0083 ms
Coefficient wise mul	1040	0.0083 ms
Gaussian sampling	1080	0.0086 ms
Inverse CRT	28	0.0002 ms
Decryption	19191	0.153 ms
Encryption	34385	0.275 ms
Recryption †	53576	0.428 ms

† The recryption-box is instantiated in the first mode (i.e. with key switching) or in the second mode (threshold sharing) with $th = 1$.

To get a sense of comparison with the actual bootstrapping operation, we consider the FHE implementation on an Intel Core i7 processor running at 3.4GHZ in [25]. The authors in [25] implement the FHE scheme over integers and choose a very large parameter set to support the bootstrapping operation. One bootstrapping operation requires 172 s to refresh encrypted data. In comparison, using the proposed single-processor recryption-box we can refresh a ciphertext in only 0.6 ms; this is roughly 2.8×10^5 times faster than the bootstrapping operation in [25]. Moreover the efficiency gain is not only restricted to the cleaning of the encrypted data. The large parameter set used in [25] also slows down the homomorphic multiplication and addition operations. One homomorphic multiplication in [25] takes 0.72 s; whereas for our parameter set it takes roughly 11 ms on a single core running at 3.1 GHZ.

7.6 Summary

In this chapter we have proposed a recryption-box model to assist fully homomorphic function evaluation. This recryption-box is used to bypass the costly bootstrapping operation and achieve an order of magnitude speedup in homomorphic evaluation time. We described two possible instantiations of the recryption-box and analyzed their security and ease of use. In our opinion, the main advantage of the recryption-box is that the costly bootstrapping mechanism is no longer required, and therefore with this we can reduce the parameters of the somewhat homomorphic encryption scheme and achieve near practical evaluation time. We demonstrated the soundness of our proposal by

implementing a recryption-box assisted encrypted search that achieves nearly twenty times speedup with respect to an implementation that does not use a recryption-box.

Chapter 8

Conclusions and future work

In this chapter we summarize the contributions of this thesis and point out some of the possible future directions.

8.1 Conclusions

Strong ECC is feasible on IoT. Efficient implementations of elliptic-curve cryptography (ECC) targeting different application requirements have received interest for over two decades. However the proposals for implementing ECC on tiny devices focused predominately on 163-bit elliptic-curves which provide only 80-bit security. Feasibility of larger elliptic-curves on such devices was not investigated. In this thesis we designed a 140-bit secure lightweight ECC architecture based on a 283-bit Koblitz curve with countermeasures against timing and power side channel attacks. When instantiated as a coprocessor of commercial 16-bit microcontrollers, the ECC architecture consumes only 4.3 KGE, showing its potential use in low-end Internet of things (IoT) devices.

Ring-LWE-based PKC is fast. Construction of public-key cryptography (PKC) primitives that are secure against quantum computers is a very recent topic. In this thesis we investigated the implementation aspects of public-key encryption based on the *ring learning with errors* (ring-LWE) problem, which is presumed to be secure against quantum computers. We analyzed the arithmetic primitives, namely discrete Gaussian sampling and polynomial arithmetic. We showed that high precision discrete Gaussian sampling can be

implemented in hardware using a very small amount of resources following an adaptation of the Knuth-Yao algorithm. Our sampler architecture is also very fast. For polynomial multiplication, we used the NTT method coupled with additional optimizations in the computation steps and the architecture. As a result of our design decisions and optimization strategies, the implemented public-key encryption processor achieves very fast computation time ($48/21\mu s$ per encryption/decryption) while using minimum area and memory.

Hardware accelerates SHE, yet not enough. When we started this research, very few publications existed on implementing homomorphic encryption schemes in hardware. We designed the first hardware architecture of the building blocks required for the ring-LWE-based homomorphic encryption scheme YASHE. To accelerate the arithmetic on large polynomials with large coefficient size, we proposed computational and architectural optimizations. For a proof of concept implementation on a Xilinx ML605 board, it turned out that the arithmetic operations can be accelerated using the FPGA; but the large data transfer which happens between the FPGA and the external memory, slows down the speed. With a more advanced memory interface and a larger FPGA, we can achieve significant speedup with respect to software implementations. But even then, the speedup will not be able to make homomorphic encryption fast enough for deployment on cloud computers.

With some trust, FHE is close to being practical. Since hardware accelerators are not fast enough, we designed a special hardware module called *recryption box* to assist homomorphic function evaluation. The security of this module is related to the security of a multiparty computation scheme; hence there is a certain amount of trust involved. We evaluated encrypted search as an example application, and observed that the recryption box can accelerate the encrypted search by a factor of twenty.

8.2 Future works

ECC-ring-LWE hybrid schemes. There are new publications [20, 19] that propose hybrid public-key schemes for present-day applications: ring-LWE for public-key encryption or key exchange and ECC for digital signature. The idea is that encryption or key exchange schemes should remain secure against future quantum computing attacks; whereas for digital signature based authentication schemes, security against the existing cryptanalysis techniques is sufficient. Hence it would be interesting to design a unified cryptoprocessor by combining

the Koblitz curve processor of Chap. 3 with with the ring-LWE-based public-key encryption processor of Chap. 5.

Post-quantum digital signature schemes. In this thesis we investigated the implementation aspects of ring-LWE-based public-key and homomorphic encryption schemes. There are several post-quantum digital signature schemes that work on polynomial rings. The main challenge in some of these schemes is that they require sampling from a discrete Gaussian distribution with large standard deviation. Pöppelmann, Ducas and Güneysu [103] showed how to efficiently sample from such wide distributions. Still it will be interesting to develop more efficient sampling algorithms that achieve better performance. New signature schemes, such as TESLA [8], do not require Gaussian sampling during signature generation. It will be interesting to evaluate their performance on hardware platforms.

Post-quantum cryptography for IoT. From this thesis we can conclude that ring-LWE problem based public-key cryptography is as computationally intensive as the classical schemes. IoT devices are constrained by the amount of available resources such as computation capability, storage or memory, power and energy consumption.

In this research, we mainly investigated the implementation aspects of ring-LWE-based post-quantum public-key cryptography and performed implementation specific optimizations targeting fast computation time. An interesting direction for future work would be to investigate lightweight design methodology taking into account the limitations of IoT devices such as small silicon area and low energy/power consumption. This would require mathematical or system-level optimizations and modifications tailored towards IoT. Ring-LWE-based cryptography relies a lot on polynomial arithmetic. Hence it would be interesting to design algorithms that could perform polynomial arithmetic by consuming a very small amount of resources.

Customized FPGAs for homomorphic schemes. In this thesis we found that the memory access overhead is the main factor in restricting the speed of homomorphic evaluation using FPGAs. Hence designing of FPGAs specific to homomorphic function evaluation will be an interesting future research. For e.g., if FPGAs are manufactured with sufficient amount of on-chip memory to store two ciphertexts, then the overhead of external memory access could be reduced significantly. Beside this, integration of a cache memory to the FPGA chip would enable prefetching of data from the slower external memory, and hence would reduce the time spent in the external communication.

Protection against physical attacks. In this thesis we developed efficient algorithms and architectures for ring-LWE-based cryptographic schemes. Designing of countermeasures against side channel and fault attacks would be a very interesting future research.

Effect of bias in randomness on Gaussian sampling. The discrete Gaussian sampler requires random numbers. Any bias in the randomness would result in a large statistical distance to the accurate Gaussian distribution, and this could be exploited by an attacker. Hence it would be interesting to study the effect of a biased random number generator on the Gaussian sampling.

Appendix A

High speed scalar conversion for Koblitz curves

Here we show that optimization tricks similar to Sect. 3.2 can be applied to design a high speed scalar conversion algorithm. We choose the high-speed variant of the lazy reduction algorithm [62] known as the *double lazy reduction*. The algorithm is based on the fact that division by τ^2 is also easy to perform in hardware following Theorem 2 in Sect. 2.2.1. During a scalar reduction, the algorithm performs repeated divisions by τ^2 for $(m-1)/2$ number of times. As a result, the cycle requirement reduces to nearly half compared to the lazy reduction. The computational steps performed in the double lazy reduction are shown in Alg. 19.

Elimination of long subtractions for nonzero remainders

In line 6 of Alg. 19, remainders u_0 and $u_1 \in \{0, 1\}$ are subtracted from d_0 and d_1. We observe that the subtractions are *easy* in some cases. For example, when $d_0 \equiv 1 \ (mod\ 4)$ and $2d_1 \equiv 0 \ (mod\ 4)$ (i.e. $u_0 = 1$ and $u_1 = 0$), the subtraction of u_0 from d_0 is equivalent to changing the least significant bit of d_0 from 1 to 0. Hence, in this case the long subtraction can be replaced by a bit alteration. However, when carry propagations are involved with long subtractions, alteration of few specific bits do not work as a replacement. For example, when $d_0 \equiv 3 \ (mod\ 4)$ and $2d_1 \equiv 0 \ (mod\ 4)$ (i.e. when $u_0 = 1$ and $u_1 = 1$), a long subtraction appears. Use of signed remainder set u_0 and $u_1 \in \{0, \pm 1\}$ helps to a certain extent in eliminating the long subtractions of

Algorithm 19: *Fast scalar reduction from [62]*

Input: integer k
Output: reduced scalar γ

```
1  begin
2      (a_0, a_1) ← (1, 0), (b_0, b_1) ← (0, 0), (d_0, d_1) ← (k, 0) ;
3      for i = 1 to (m − 1)/2 do
4          u ← (d_0 − 2d_1) mod 4 ;
5          u_0 ← u mod 2, u_1 ← ⌊u/2⌋ ;
6          d_0 ← d_0 − u_0, d_1 ← d_1 − u_1 ;
7          (d_0, d_1) ← ((−d_0 − 2d_1)/4, −(−d_0 + 2d_1)/4) ;
8          if u > 0 then
9              b_0 ← (b_0 + u_0 a_0 − 2u_1 a_1) ;
10             b_1 ← b_1 + u_0 a_1 + u_1 (a_0 − a_1) ;
11         end
12         (a_0, a_1) ← (−2(a_0 − a_1), −a_0 − a_1) ;
13     end
14     if d_0 ≡ 1(mod 2) then
15         u ← 1, d_0 ← d_0 − 1 ;
16         (b_0, b_1) ← (b_0 + a_0, b_1 + a_1) ;
17     end
18     (d_0, d_1) ← (−d_0/2 + d_1, −d_0/2) ;
19     γ ← (b_0 + d_0, b_1 + d_1) ;
20 end
```

nonzero remainders for such cases. Table A.1 shows how the signed remainders are generated during the reduction steps depending on the low bits of d_0 and $2d_1$. It can be noticed from the table that, except Case 4, subtractions of the u_0 and u_1 from d_0 and d_1 involve no carry propagation and thus can be replaced by alterations of the low bits in d_0 and d_1.

For Case 4, if we perform the subtraction of $u_0 = -1$ in line 7 of Alg. 19 instead of line 6 (i.e. we put $d_0 + 1$ in place of d_0), then we have the following observation.

$$d_0 \leftarrow -\frac{2d_1 + (d_0 + 1)}{4}$$

$$d_1 \leftarrow -\frac{2d_1 - (d_0 + 1)}{4}$$

Table A.1: Signed remainders during reduction of scalar

Cases	$d_0\ (mod\ 4)$	$2d_1\ (mod\ 4)$	u_0	u_1
1	0	0	0	0
2	1	0	1	0
3	2	0	0	-1
4	3	0	-1	0
5	0	2	0	1
6	1	2	-1	0
7	2	2	0	0
8	3	2	1	0

This is equivalent to taking carry/borrow inputs in the adder/subtracter circuits during the computations of d_0 and d_1. This is shown below.

$$d_0 \leftarrow -\frac{2d_1 + d_0 + (Carry\ input\ =\ 1)}{4}$$

$$d_1 \leftarrow -\frac{2d_1 - d_0 - (Borrow\ input\ =\ 1)}{4}$$

From the observations presented in this subsection, we draw the conclusion that the long subtractions of the nonzero remainders can be eliminated by changing the low order bits of d_0 and d_1 or by considering carry/borrow inputs to the adder/subtracter circuits.

Elimination of subtractions from zero

In line 7 of Alg. 19, a subtraction from zero is required for d_0 after computing $2d_1 + d_0$ (Eq. A.1).

$$(d_0, d_1) \leftarrow (-\frac{2d_1 + d_0}{4}, -\frac{2d_1 - d_0}{4}) \tag{A.1}$$

We eliminate the subtraction from zero using the following scheme. Instead of Eq. A.1, we compute Eq. A.2.

$$(d_0, d_1) \leftarrow (\frac{2d_1 + d_0}{4}, \frac{2d_1 - d_0}{4}) \tag{A.2}$$

The results from Eq. A.1 and A.2 have opposite signs, but same magnitudes. So, when Eq. A.1 and A.2 are computed inside the for-loop in Alg. 19 for

an even number of times, the results of the equations are same; otherwise the results have opposite signs.

The same trick is applied to eliminate the subtractions from zero during the computation of (a_0, a_1) in line 12 of Alg. 19. Instead of computing Eq. A.3

$$(a_0, a_1) \leftarrow (-2(a_0 - a_1),\ -(a_0 + a_1)) \tag{A.3}$$

we compute Eq. A.4.

$$(a_0, a_1) \leftarrow (2(a_0 - a_1),\ (a_0 + a_1)) \tag{A.4}$$

After completion of the for-loop, one subtraction from zero is required to make the signs correct for (a_0, a_1) when $(m-1)/2$ is odd. This subtraction from zero can be eliminated if we compute Eq. A.5 in line 16 of Alg. 19.

$$(b_0, b_1) \leftarrow (b_0 - a_0, b_1 - a_1) \tag{A.5}$$

The improved high-speed scalar reduction algorithm is described in Alg. 20.

Algorithm 20: *New Reduction Algorithm*

Input: integer k
Output: reduced scalar γ

begin
  $(a_0, a_1) \leftarrow (1, 0),\ (b_0, b_1) \leftarrow (0, 0),\ (d_0, d_1) \leftarrow (k, 0)$;
  /* Iterative divisions by τ^2 start here */ ;
  for $i = 1$ *to* $(m-1)/2$ **do**
    $u \leftarrow (d_0 - 2d_1)\ mod\ 4$;
    $(u_0, u_1) \leftarrow Table\ 1$;
    $(d_0, d_1) \leftarrow AlterLowBits(d_0, d_1)$;
    if $Case\ 4\ is\ True$ **then**
      $(B, C) \leftarrow (1, 1)$ /* Borrow and Carry Inputs */
    end
    else
      $(B, C) \leftarrow (0, 0)$
    end
    $(d_0, d_1) \leftarrow ((2d_1 + d_0 + C)/4, (2d_1 - d_0 - B)/4)$;
    if $u > 0$ **then**
      $b_0 \leftarrow (b_0 + u_0 a_0 - 2u_1 a_1)$;
      $b_1 \leftarrow (b_1 + u_0 a_1 + u_1(a_0 - a_1))$;
    end
    $(a_0, a_1) \leftarrow (2(a_0 - a_1),\ a_0 + a_1)$;
  end
  /* Iterative divisions by τ^2 finish here */ ;
  if $d_0 \equiv 1 (mod\ 2)$ **then**
    $d_0 \leftarrow AlterLeastBit(d_0)$;
    if $\frac{m-1}{2} \equiv 0 (mod\ 2)$ **then**
      $(b_0, b_1) \leftarrow (b_0 + a_0, b_1 + a_1)$;
    end
    else
      $(b_0, b_1) \leftarrow (b_0 - a_0, b_1 - a_1)$;
    end
  end
  $(d_0, d_1) \leftarrow ((2d_1 - d_0)/2, d_0/2)$ /* Final division by τ */ ;
  if $\frac{m-1}{2} \equiv 0 (mod\ 2)$ **then**
    $\gamma \leftarrow (b_0 + d_0, b_1 - d_1)$;
  end
  else
    $\gamma \leftarrow (b_0 - d_0, b_1 + d_1)$;
  end
end

A.1 Improved double digit τNAF generation

In [62], two consecutive τNAF digits are generated in a single step from the reduced scalar $d_0 + \tau d_1$ by performing divisions by τ^2. The authors call the NAF as *double τNAF*. Table A.2 shows how the consecutive τNAF digits r_0 and r_1 are generated by observing the low order bits of d_0 and d_1. Similar to Section 3.2.1, we eliminate the subtractions of nonzero remainders from d_0 and d_1 during the τNAF generation process.

From Table A.2, we see that for the cases 2, 3.B, 3.C, 3.D, 5.A, 5.B, 5.D, 6 and

Table A.2: NAF Generation for $\mu = -1$

Cases	$d_0(mod4)$	$2d_1(mod4)$	$d_0(mod8)$	$2d_1(mod8)$	r_0	r_1
1	0	0			0	0
2	1	0			1	0
3.A	2	0	2	0	0	1
3.B	2	0	6	0	0	-1
3.C	2	0	2	4	0	-1
3.D	2	0	6	4	0	1
4	3	0			-1	0
5.A	0	2	0	2	0	1
5.B	0	2	4	2	0	-1
5.C	0	2	0	6	0	-1
5.D	0	2	4	6	0	1
6	1	2			-1	0
7	2	2			0	0
8	3	2			1	0

8, the subtractions of nonzero remainders from d_0 or d_1 affect only the low order bits of d_0 and d_1. For the above cases, the long subtractions are replaced by cheaper bit alterations in d_0 and d_1. Subtraction of $r_0 = -1$ in Case 4 in Table A.2 can be handled in the same way we did for Case 4 in Table A.1 (Sect. 20).

In Case 3.A, the subtraction of $r_1 = 1$ from d_1 involves borrow propagation and thus may affect all the bits of d_1. If we incorporate this subtraction in the next step where we perform the division by τ^2, then by putting $d_1 - 1$ in place of d_1 in Eq. A.1, we have the following observation.

$$\begin{aligned}(d_0, d_1) &\leftarrow (-\frac{2(d_1-1)+d_0}{4}, -\frac{2(d_1-1)-d_0}{4}) \\ &\leftarrow (-\frac{2d_1+(d_0-2)}{4}, -\frac{2d_1-(d_0+2)}{4}) \end{aligned} \tag{A.6}$$

Thus we find that the subtraction of r_1 from d_1 is equivalent to the addition or subtraction of two with d_0. As $d_0 \equiv 2 \ (mod\ 8)$, the subtraction or addition of 2 changes only the three low bits of d_0.

In Case 5.C, the subtraction of $r_1 = -1$ from d_1 involves carry propagation. When we put $d_1 + 1$ in place of d_1 in Eq. A.1, we have the following observation.

$$\begin{aligned}(d_0, d_1) &\leftarrow (-\frac{2(d_1+1)+d_0}{4}, -\frac{2(d_1+1)-d_0}{4}) \\ &\leftarrow (\frac{2\bar{d_1}-d_0}{4}, \frac{2\bar{d_1}+d_0}{4}) \end{aligned} \tag{A.7}$$

Algorithm 21: *New τNAF Generation Algorithm*

Input: Reduced Scalar $\gamma = d_0 + \tau d_1$
Output: $\tau\text{NAF}(\gamma)$

begin
  $S \leftarrow \langle\rangle$ /* Used to store τNAF */ ;
  $Sign \leftarrow 0$ /* Used to keep sign of (d_0, d_1) */ ;
  while $d_0 \neq 0$ *or* $d_1 \neq 0$ **do**
    $(r_0, r_1) \leftarrow$ *Table* 2 ;
    $(d_0, d_1) \leftarrow AlterLowBits(d_0, d_1)$ *in Sect. A.*1 ;
    $(B, C) \leftarrow (0, 0)$ /* Borrow and Carry Inputs */ ;
    if $Sign = 1$ **then**
      $(r_0, r_1) \leftarrow (-r_0, -r_1)$;
    end
    Prepend (r_1, r_0) *to* S /* τNAF digits */ ;
    if *Case* 4 *True* **then**
      $(B, C) \leftarrow (1, 1)$
    end
    if *Case* 5.*C True* **then**
      $(d_0, d_1) \leftarrow (\frac{2\bar{d}_1 - d_0}{4}, \frac{2\bar{d}_1 + d_0}{4})$;
      $Sign \leftarrow Sign$;
    end
    else
      $(d_0, d_1) \leftarrow (\frac{2d_1 + d_0 + C}{4}, \frac{2d_1 - d_0 - B}{4})$;
      $Sign \leftarrow 1 \oplus Sign$;
    end
  end
end

So, using one's complement of d_1 in Case 5.C, we eliminate the long subtraction of r_1 from d_1. Computing one's complement in hardware platform is easy as all the bits of d_1 can be altered in parallel.

Algorithm 21 describes the steps of the new τNAF generation technique. Only one addition or subtraction operations are performed on d_0 and d_1 during division by τ^2 in any iteration. Thus, for the τNAF generation part of the scalar conversion, presence of only one adder/subtracter circuits in the critical paths of d_0 and d_1 is sufficient.

A.2 Hardware architecture

We use these optimizations to design a high-speed and pipelined scalar conversion architecture. The architecture is described in details our publication

Sujoy Sinha Roy, Junfeng Fan, Ingrid Verbauwhede. Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23,*

no. 5, pp. 810-818, May 2015.

We would like to mention that the high-speed architecture is a proof of concept implementation and is not resistant against side channel attacks.

Table A.3: Performance results on Xilinx Virtex 4 FPGA

Work	Slices	Freq MHz	Reduction Time (μs)	Conversion Time (μs)
Brumley et. al.[62]	1671	65.9	4.3	8.6
Adikari et. al.[62]	1998	65.1	2.2	4.4
Our implementation	1814	107	1.3	2.6

A.3 Implementation results

Table A.3 shows the performance of our high-speed scalar conversion architecture for K-283. Results were obtained from Xilinx ISEv12.2 tool after place and route analysis with optimization for speed. The conversion time is the total time required for the scalar reduction and the complete τNAF generation.

Appendix B

Implementation of operations used by algorithm 6

The operations required by Alg. 6 are implemented by combining an FSM and a program ROM. The program ROM includes subprograms for all operations of Alg. 6 and the FSM sets the address of the ROM to the first instruction of the subprogram according the phase of the algorithm and t_{i+1}, t_i.

Table B.1 shows the contents of the program ROM. The operations required by Alg. 6 are in this ROM as follows:

- Line 0 obtains the next bits of the zero-free representation.
- Lines 1–23 perform the precomputation that computes $(x_{+1}, y_{+1}) = \phi(P) + P$ and $(x_{-1}, y_{-1}) = \phi(P) - P$.
- Line 24 computes the negative of Q during the initialization and Line 25 is the corresponding dummy operation.
- Lines 26–28 randomize the projective coordinates of Q by using the random $r \in \mathbb{F}_{2^{283}}$ which is stored in Z.
- Lines 29–34 compute two Frobenius endomorphisms for Q.
- Lines 35–37 set $(x_p, y_p) \leftarrow (x_{+1}, y_{+1}) = \phi(P) + P$ and compute the y-coordinate of its negative to y_m.
- Lines 38–40 set $(x_p, y_p) \leftarrow (x_{-1}, y_{-1}) = \phi(P) - P$ and compute the y-coordinate of its negative to y_m.

Table B.1: The program ROM includes instructions for the following operations

0 Convert(k)	**20** $y_{-1} \leftarrow y_{-1} \times T_1$	**40** $y_m \leftarrow x_{-1} + y_{-1}$	**60** $T_2 \leftarrow x_p \times Z$
1 $x_{+1} \leftarrow X^2$	**21** $y_{-1} \leftarrow y_{-1} + x_{-1}$	**41** $x_p \leftarrow x_{+1}$	**61** $T_2 \leftarrow T_2 + X$
2 $y_{+1} \leftarrow Y^2$	**22** $y_{-1} \leftarrow y_{-1} + Y$	**42** $y_m \leftarrow y_{+1}$	**62** $Y \leftarrow Y + Z$
3 $x_{+1} \leftarrow X + x_{+1}$	**23** $y_{-1} \leftarrow y_{-1} + X$	**43** $y_p \leftarrow x_{+1} + y_{+1}$	**63** $Y \leftarrow Y \times T_2$
4 $x_{-1} \leftarrow x_{+1}^{-1}$	**24** $Y \leftarrow X + Y$	**44** $x_p \leftarrow x_{-1}$	**64** $T_1 \leftarrow Z^2$
5 $T_1 \leftarrow Y + y_{+1}$	**25** $T_1 \leftarrow X + Y$	**45** $y_m \leftarrow y_{-1}$	**65** $T_1 \leftarrow T_1 \times y_m$
6 $y_{-1} \leftarrow T_1 \times x_{-1}$	**26** $X \leftarrow X \times Z$	**46** $y_p \leftarrow x_{-1} + y_{-1}$	**66** $Y \leftarrow Y + T_1$
7 $T_1 \leftarrow y_{-1}^2$	**27** $T_1 \leftarrow Z^2$	**47** $T_1 \leftarrow Z^2$	**67** $x_{+1} \leftarrow Z$
8 $T_1 \leftarrow T_1 + y_{-1}$	**28** $Y \leftarrow Y \times T_1$	**48** $T_1 \leftarrow T_1 \times y_p$	**68** $x_{-1} \leftarrow x_{+1}^{-1}$
9 $x_{+1} \leftarrow T_1 + x_{+1}$	**29** $Y \leftarrow Y^2$	**49** $T_1 \leftarrow T_1 + Y$	**69** $X \leftarrow X \times x_{-1}$
10 $T_1 \leftarrow x_{+1} + X$	**30** $Y \leftarrow Y^2$	**50** $T_2 \leftarrow Z \times x_p$	**70** $x_{-1} \leftarrow x_{-1}^2$
11 $y_{+1} \leftarrow T_1 + y_{-1}$	**31** $X \leftarrow X^2$	**51** $T_2 \leftarrow T_2 + X$	**71** $Y \leftarrow Y \times x_{-1}$
12 $y_{+1} \leftarrow y_{+1} + x_{+1}$	**32** $X \leftarrow X^2$	**52** $X \leftarrow T_2^2$	**72** $X \leftarrow x_{+1}$
13 $y_{+1} \leftarrow y_{+1} + Y$	**33** $Z \leftarrow Z^2$	**53** $X \leftarrow X + T_1$	**73** $Y \leftarrow y_{+1}$
14 $x_{-1} \leftarrow x_{-1} \times X$	**34** $Z \leftarrow Z^2$	**54** $T_2 \leftarrow T_2 \times Z$	**74** $X \leftarrow x_{-1}$
15 $y_{-1} \leftarrow y_{-1} + x_{-1}$	**35** $x_p \leftarrow x_{+1}$	**55** $X \leftarrow X \times T_2$	**75** $Y \leftarrow y_{-1}$
16 $T_1 \leftarrow x_{-1}^2$	**36** $y_p \leftarrow y_{+1}$	**56** $Y \leftarrow T_1 \times T_2$	**76** $T_1 \leftarrow x_{+1}$
17 $x_{-1} \leftarrow x_{-1} + T_1$	**37** $y_m \leftarrow x_{+1} + y_{+1}$	**57** $T_1 \leftarrow T_1^2$	**77** $T_2 \leftarrow y_{+1}$
18 $x_{-1} \leftarrow x_{-1} + x_{+1}$	**38** $x_p \leftarrow x_{-1}$	**58** $X \leftarrow X + T_1$	
19 $T_1 \leftarrow x_{-1} + X$	**39** $y_p \leftarrow y_{-1}$	**59** $Z \leftarrow T_2^2$	

- Lines 41–43 compute $(x_p, y_p) \leftarrow -(x_{+1}, y_{+1}) = -\phi(P) - P$ and set the y-coordinate of its negative to y_m.
- Lines 44–46 compute $(x_p, y_p) \leftarrow -(x_{-1}, y_{-1}) = -\phi(P) + P$ and set the y-coordinate of its negative to y_m.
- Lines 47–66 compute the point addition $(X, Y, Z) \leftarrow (X, Y, Z) + (x_p, y_p)$ in López-Dahab coordinates using the equations from [5].
- Lines 67–71 recover the affine coordinates of Q by computing $(X, Y) \leftarrow (X/Z, Y/Z^2)$.
- Lines 72–73 and lines 74–75 initialize Q with (x_{+1}, y_{+1}) and (x_{-1}, y_{-1}), respectively, and lines 76–77 perform a dummy operation for these operations.

Point addition and point subtraction are computed with exactly the same sequence of operations. This is achieved by introducing an initialization which sets the values of three internal variables x_p, y_p, and y_m according to Table B.2

(these are in lines 35–46 in Table B.1). This always requires two copy instructions followed by an addition. After this initialization, both point addition and point subtraction are computed with a common sequence of operations which adds the point (x_p, y_p) to Q. The element x_m is the y-coordinate of the negative of (x_p, y_p) and it is also used during the point addition.

Table B.2: Initialization of point addition and point subtraction

t_{i+1}, t_i	1st	2nd	3rd
$+1, +1$	$x_p \leftarrow x_{+1}$	$y_p \leftarrow y_{+1}$	$y_m \leftarrow x_{+1} + y_{+1}$
$+1, -1$	$x_p \leftarrow x_{-1}$	$y_p \leftarrow y_{-1}$	$y_m \leftarrow x_{-1} + y_{-1}$
$-1, +1$	$x_p \leftarrow x_{-1}$	$y_m \leftarrow y_{-1}$	$y_p \leftarrow x_{-1} + y_{-1}$
$-1, -1$	$x_p \leftarrow x_{+1}$	$y_m \leftarrow y_{+1}$	$y_p \leftarrow x_{+1} + y_{+1}$

Bibliography

[1] M. Abdalla, M. Bellare, and P. Rogaway. The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES. In *Topics in Cryptology — CT-RSA 2001: The Cryptographers' Track at RSA Conference 2001 San Francisco, CA, USA, April 8–12, 2001 Proceedings*, pages 143–158, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[2] M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.

[3] M. Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 10–19, New York, NY, USA, 1998. ACM.

[4] M. Ajtai, R. Kumar, and D. Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 601–610, New York, NY, USA, 2001. ACM.

[5] E. Al-Daoud, R. Mahmod, M. Rushdan, and A. Kilicman. A New Addition Formula for Elliptic Curves over $GF(2^n)$. *IEEE Transactions on Computers*, 51(8):972–975, Aug. 2002.

[6] M. Albrecht, S. Bai, and L. Ducas. *A Subfield Lattice Attack on Overstretched NTRU Assumptions*, pages 153–178. Advances in Cryptology – CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[7] M. R. Albrecht. Complexity Estimates for Solving LWE. `https://bitbucket.org/malb/lwe-estimator/raw/HEAD/estimator.py`.

[8] E. Alkim, N. Bindel, J. Buchmann, Özgür Dagdelen, and P. Schwabe. TESLA: Tightly-Secure Efficient Signatures from Standard Lattices. Cryptology ePrint Archive, Report 2015/755, 2015. http://eprint.iacr.org/2015/755.

[9] D. F. Aranha, R. Dahab, J. López, and L. B. Oliveira. Efficient Implementation of Elliptic Curve Cryptography in Wireless Sensors. *Advances in Mathematics of Communications*, 4(2):169–187, 2010.

[10] A. Aysu, C. Patterson, and P. Schaumont. Low-Cost and Area-Efficient FPGA Implementations of Lattice-Based Cryptography. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 81–86, June 2013.

[11] R. Azarderakhsh, K. U. Järvinen, and M. Mozaffari-Kermani. Efficient Algorithm and Architecture for Elliptic Curve Cryptography for Extremely Constrained Secure Applications. *IEEE Transactions on Circuits and Systems I—Regular Papers*, 61(4):1144–1155, Apr. 2014.

[12] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for Key Management – Part 1: General. http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf, March, 2007, Revised January 2016.

[13] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Low-Cost Elliptic Curve Cryptography for Wireless Sensor Networks. In *Security and Privacy in Ad-Hoc and Sensor Networks — ESAS 2006*, volume 4357 of *Lecture Notes in Computer Science*, pages 6–17. Springer, 2006.

[14] BBC News. NSA 'Developing Code-Cracking Quantum Computer'. January 2014. http://www.bbc.com/news/technology-25588605.

[15] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Lightweight Block Ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[16] D. Bernstein. Fast Multiplication and its Applications. *Algorithmic Number Theory*, 44:325–384, 2008.

[17] D. J. Bernstein, S. Engels, T. Lange, R. Niederhagen, C. Paar, P. Schwabe, and R. Zimmermann. Faster Elliptic-Curve Discrete Logarithms on FPGAs. Cryptology ePrint Archive, Report 2016/382, 2016. http://eprint.iacr.org/2016/382.

[18] H. Bock, M. Braun, M. Dichtl, E. Hess, J. Heyszl, W. Kargl, H. Koroschetz, B. Meyer, and H. Seuschek. A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. In *Proceedings of the 4th Workshop on RFID Security — RFIDSec 2008*, 2008.

[19] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1006–1018, New York, NY, USA, 2016. ACM.

[20] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570, May 2015.

[21] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In *Proceedings of the 14th IMA International Conference on Cryptography and Coding (IMACC 2013)*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013.

[22] Z. Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology — CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[23] B. B. Brumley and K. U. Järvinen. Conversion Algorithms and Implementations for Koblitz Curve Cryptography. *IEEE Transactions on Computers*, 59(1):81–92, Jan. 2010.

[24] J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden. Discrete Ziggurat: A Time-Memory Trade-Off for Sampling from a Gaussian Distribution over the Integers. In *Selected Areas in Cryptography – SAC 2013: 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 402–417, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[25] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun. *Batch Fully Homomorphic Encryption over the Integers*, pages 315–335. Advances in Cryptology – EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[26] A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N. P. Smart. Between a Rock and a Hard Place: Interpolating between MPC and FHE. In *Advances in Cryptology - ASIACRYPT 2013: 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, pages 221–240, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[28] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[29] J.-S. Coron. Resistance Against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems — CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[30] J.-S. Coron, T. Lepoint, and M. Tibouchi. Scale-Invariant Fully Homomorphic Encryption over the Integers. In *Public-Key Cryptography — PKC 2014*, volume 8383 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2014.

[31] R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient Software Implementation of Ring-LWE Encryption. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 339–344, 2015.

[32] R. de Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede. Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 112:1–112:6, New York, NY, USA, 2014. ACM.

[33] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[34] M. Dichtl and J. D. Golic. High-Speed True Random Number Generation with Logic Gates Only. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *LNCS*, pages 45–62. Springer Berlin, 2007.

[35] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[36] Y. Doröz, E. Öztürk, E. Savas, and B. Sunar. Accelerating LTV Based Homomorphic Encryption in Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International*

Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings, pages 185–204, 2015.

[37] Y. Doröz, E. Öztürk, and B. Sunar. Evaluating the Hardware Performance of a Million-bit Multiplier. In *Proceedings of the 16th Euromicro Conference on Digital System Design (DSD 2013)*, pages 955–962, 2013.

[38] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. *Lattice Signatures and Bimodal Gaussians*, pages 40–56. Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[39] N. Dwarakanath and S. Galbraith. Sampling from Discrete Gaussians for Lattice-Based Cryptography on a Constrained Device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, 2014.

[40] M. D. Ercegovac and T. Lang. Chapter 1 - Review of Basic Number Representations and Arithmetic Algorithms. In *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 3 – 49. Morgan Kaufmann, San Francisco, 2004.

[41] M. D. Ercegovac and T. Lang. Chapter 7 - Reciprocal, Division, Reciprocal Square Root, and Square Root by Iterative Approximation. In *Digital Arithmetic*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 366 – 395. Morgan Kaufmann, San Francisco, 2004.

[42] J. Fan and I. Verbauwhede. An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost. In *Cryptography and Security: From Theory to Applications*, volume 6805 of *Lecture Notes in Computer Science*, pages 265–282. Springer, 2012.

[43] J. Fan and F. Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `http://eprint.iacr.org/`.

[44] P.-A. Fouque and F. Valette. The Doubling Attack—Why Upwards is Better than Downwards. In *Cryptographic Hardware and Embedded Systems — CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2003.

[45] S. D. Galbraith and P. Gaudry. Recent Progress on the Elliptic Curve Discrete Logarithm Problem. *Des. Codes Cryptography*, 78(1):51–72, Jan. 2016.

[46] C. Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC 2009)*, pages 169–178, 2009.

[47] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology — CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[48] C. Gentry and Z. Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In *Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005. Proceedings*, pages 803–815, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[49] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology — CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[50] I. Goldberg. Improving the Robustness of Private Information Retrieval. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 131–148, May 2007.

[51] J. D. Golic. New Methods for Digital Generation and Postprocessing of Random Data. *IEEE Transactions on Computers*, 55(10):1217–1229, 2006.

[52] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer Berlin, 2012.

[53] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 323–345, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[54] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[55] D. Hein, J. Wolkerstorfer, and N. Felber. ECC Is Ready for RFID – A Proof in Silicon. In *Selected Areas in Cryptography — SAC 2008*, volume 5381 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2009.

[56] G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar. Full-Size High-Security ECC Implementation on MSP430 Microcontrollers. In *Progress in Cryptology — LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 31–47. Springer, 2015.

[57] P. Hirschhorn, J. Hoffstein, N. Howgrave-graham, and W. Whyte. Choosing NTRUEncrypt Parameters in Light of Combined Lattice Reduction and MITM Approaches. In *In Proc. ACNS 2009, LNCS 5536*, pages 437–455. Springer-Verlag, 2009.

[58] J. Hoffstein, J. Pipher, and J. Silverman. *An Introduction to Mathematical Cryptography*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[59] N. Howgrave Graham. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU. In *Advances in Cryptology - CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 150–169. Springer Berlin Heidelberg, 2007.

[60] Intel®. Core™i7-2600 processor. `https://ark.intel.com/products/52213/Intel-Core-i7-2600-Processor-8M-Cache-up-to-3_80-GHz`.

[61] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Information and Computation*, 78(3):171–177, Sept. 1988.

[62] J. Adikari, V.S. Dimitrov, and K. Järvinen. A Fast Hardware Architecture for Integer to τNAF Conversion for Koblitz Curves. *Computers, IEEE Transactions on*, 61(5):732 –737, may 2012.

[63] A. Kamal and A. Youssef. An FPGA Implementation of the NTRUEncrypt Cryptosystem. In *Microelectronics (ICM), 2009 International Conference on*, pages 209–212, Dec 2009.

[64] A. Kargl, S. Pyka, and H. Seuschek. Fast Arithmetic on ATmega128 for Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2008/442, 2008.

[65] A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Constant-time Discrete Gaussian Sampling. In *under review*, 2017.

[66] A. H. Karp and P. Markstein. High-precision Division and Square Root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997.

[67] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[68] D. E. Knuth and A. C. Yao. The Complexity of Non-Uniform Random Number Generation. *Algorithms and Complexity*, pages 357–428, 1976.

[69] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[70] N. Koblitz. CM-curves with good cryptographic properties. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, 1991.

[71] C. K. Koç, editor. *Cryptographic Engineering*. Springer, 2009.

[72] K.U. Järvinen, J. Forsten, and J.O. Skyttä. Efficient Circuitry for Computing τ-adic Non-Adjacent Form. *Proc. IEEE Int'l Conf. Electronics, Circuits and Systems (ICECS '06)*, pages 232–235, 2006.

[73] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA — A Cost-Optimized Parallel Code Breaker. In *Cryptographic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.

[74] Y. K. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-Curve-Based Security Processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, Nov. 2008.

[75] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring Polynomials with Rational Coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[76] T. Lepoint and M. Naehrig. *A Comparison of the Homomorphic Encryption Schemes FV and YASHE*, pages 318–335. Progress in Cryptology – AFRICACRYPT 2014: 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings, Springer International Publishing, Cham, 2014.

[77] T. Lepoint and M. Naehrig. A Comparison of the Homomorphic Encryption Schemes FV and YASHE. In *Progress in Cryptology — AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2014.

[78] R. Lindner and C. Peikert. Better Key Sizes (and Attacks) for LWE-based Encryption. *CT-RSA 2011*, pages 319–339, 2011.

[79] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *Information Security: 8th International Conference, ISC 2005, Singapore, September 20-23, 2005. Proceedings*, pages 314–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[80] M. Liu and P. Q. Nguyen. Solving BDD by Enumeration: An Update. In *Proceedings of the 13th International Conference on Topics in Cryptology*, CT-RSA'13, pages 293–309, Berlin, Heidelberg, 2013. Springer-Verlag.

[81] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient Ring-LWE Encryption on 8-Bit AVR Processors. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 663–682, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[82] J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in $GF(2^n)$. In *Selected Areas in Cryptography — SAC'98*, volume 1556 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 1999.

[83] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, pages 1219–1234, New York, NY, USA, 2012. ACM.

[84] V. Lyubashevsky. Lattice Signatures without Trapdoors. In *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, pages 738–755, Berlin, 2012. Springer-Verlag.

[85] V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2010.

[86] Y. Ma and L. Wanhammar. A Hardware Efficient Control of Memory Addressing for High-performance FFT Processors. *Signal Processing, IEEE Transactions on*, 48(3):917–921, Mar 2000.

[87] C. A. Melchor and P. Gaborit. A Fast Private Information Retrieval Protocol. In *2008 IEEE International Symposium on Information Theory*, pages 1848–1852, July 2008.

[88] D. Micciancio. The Hardness of the Closest Vector Problem with Preprocessing. *IEEE Trans. Information Theory*, 47(3):1212–1215, 2001.

[89] V. Miller. Uses of Elliptic Curves in Cryptography. *Advances in Cryptology, Crypto'85*, 218:417–426, 1986.

[90] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao. Targeting FPGA DSP Slices for a Large Integer Multiplier for Integer Based FHE. In *Financial Cryptography and Data Security Workshops,*

the 1st Workshop on Applied Homomorphic Cryptography and Encrypted Computing (WAHC 2013), volume 7862 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2013.

[91] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can Homomorphic Encryption Be Practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW 2011)*, pages 113–124. ACM, 2011.

[92] National Institute of Standard and Technology. FIPS 186-2, Digital Signature Standard, Federal Information Processing Standards Publication, 2000.

[93] National Institute of Standards and Technology. Discussion Paper: the Transitioning of Cryptographic Algorithms and Key Sizes. `http://csrc.nist.gov/groups/ST/key_mgmt/documents/Transitioning_CryptoAlgos_070209.pdf`, 2009.

[94] National Institute of Standards and Technology. Digital signature standard (DSS). Federal Information Processing Standard, FIPS PUB 186-4, July 2013.

[95] National Institute of Standards and Technology. NIST Kicks Off Effort to Defend Encrypted Data from Quantum Computer Threat, 2016. www.nist.gov/news-events/news/2016/04/nist-kicks-effort-defend-encrypted-data-quantum-computer-threat.

[96] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-Secure and Masked Ring-LWE Implementation. Cryptology ePrint Archive, Report 2016/1109, 2016. `http://eprint.iacr.org/2016/1109`.

[97] K. Okeya, T. Takagi, and C. Vuillaume. Efficient Representations on Koblitz Curves with Resistance to Side Channel Attacks. In *Proc. the 10th Australasian Conference on Information Security and Privacy — ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 218–229. Springer, 2005.

[98] A. Park and D. G. Han. Chosen Ciphertext Simple Power Analysis on Software 8-bit Implementation of Ring-LWE Encryption. In *2016 IEEE Asian Hardware-Oriented Security and Trust (AsianHOST)*, pages 1–6, Dec 2016.

[99] P. Pessl. Analyzing the Shuffling Side-Channel Countermeasure for Lattice-Based Signatures. In *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 153–170, Cham, 2016. Springer International Publishing.

[100] P. Pessl and M. Hutter. Curved Tags — A Low-Resource ECDSA Implementation tailored for RFID. In *Workshop on RFID Security — RFIDSec 2014*, 2014.

[101] J. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25:365–374, 1971.

[102] J. M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[103] T. Pöppelmann, L. Ducas, and T. Güneysu. *Enhanced Lattice-Based Signatures on Reconfigurable Hardware*, pages 353–370. Cryptographic Hardware and Embedded Systems – CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[104] T. Pöppelmann and T. Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158. Springer Berlin, 2012.

[105] T. Pöppelmann and T. Güneysu. Area Optimization of Lightweight Lattice-based Encryption on Reconfigurable Hardware. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2796–2799, June 2014.

[106] T. Pöppelmann and T. Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography – SAC 2013*, Lecture Notes in Computer Science, pages 68–85. Springer Berlin Heidelberg, 2014.

[107] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias. Accelerating Homomorphic Evaluation on Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 143–163, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[108] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[109] C. Rebeiro and D. Mukhopadhyay. Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*, pages 706–711, Washington, DC, USA, 2008. IEEE Computer Society.

[110] C. Rebeiro, S. S. Roy, and D. Mukhopadhyay. Pushing the Limits of High-Speed $GF(2^m)$ Elliptic Curve Scalar Multiplication on FPGAs. In *Cryptographic Hardware and Embedded Systems – CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pages 494–511, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[111] O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.

[112] O. Regev. Lattices in Computer Science. *Lecture notes of a course given in Tel Aviv University.*, 2009. `http://www.cims.nyu.edu/~regev/teaching/lattices_fall_2009/`.

[113] O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *J. ACM*, 56(6), 2009.

[114] O. Reparaz, R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Additively Homomorphic Ring-LWE Masking. In *Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, pages 233–244, Cham, 2016. Springer International Publishing.

[115] O. Reparaz, S. S. Roy, R. de Clercq, F. Vercauteren, and I. Verbauwhede. Masking Ring-LWE. *Journal of Cryptographic Engineering*, 6(2):139–153, 2016.

[116] O. Reparaz, S. S. Roy, F. Vercauteren, and I. Verbauwhede. A Masked Ring-LWE Implementation. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 683–702, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[117] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[118] S. S. Roy, J. Fan, and I. Verbauwhede. Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms. *IEEE*

Transactions on Very Large Scale Integration (VLSI) Systems, 23(5):810–818, May 2015.

[119] S. S. Roy, K. Järvinen, and I. Verbauwhede. Lightweight Coprocessor for Koblitz Curves: 283-Bit ECC Including Scalar Conversion with only 4300 Gates. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 102–122, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[120] S. S. Roy, K. Jarvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede. Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation. Cryptology ePrint Archive, Report 2015/337, 2015. `http://eprint.iacr.org/`.

[121] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede. Compact and Side Channel Secure Discrete Gaussian Sampling. Cryptology ePrint Archive, Report 2014/591, 2014. `http://eprint.iacr.org/`.

[122] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE Cryptoprocessor. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer Berlin Heidelberg, 2014.

[123] S. S. Roy, F. Vercauteren, and I. Verbauwhede. High Precision Discrete Gaussian Sampling on FPGAs. In *Selected Areas in Cryptography – SAC 2013*, Lecture Notes in Computer Science, pages 383–401. Springer Berlin Heidelberg, 2014.

[124] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2nd edition, 2013.

[125] C. P. Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Mathematical Programming*, 66(1):181–199, 1994.

[126] P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, USA, 1994. IEEE Computer Society.

[127] N. Smart and F. Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *Public Key Cryptography – PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer Berlin Heidelberg, 2010.

[128] N. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.

[129] J. A. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19(2–3):195–249, 2000.

[130] D. Stehlé and R. Steinfeld. Making NTRU as Secure as Worst-Case Problems over Ideal Lattices. In *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer Berlin Heidelberg, 2011.

[131] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In *European Conference on Wireless Sensor Networks — ESWN 2008*, volume 4913 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2008.

[132] I. T. U. Telecommunication Development Bureau. ICT Facts and Figures. `https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf`, 2015.

[133] Texas Instruments. MSP430F261x and MSP430F241x, Jun. 2007, Rev. Nov. 2012. `http://www.ti.com/lit/ds/symlink/msp430f2618.pdf` (accessed July. 22, 2015).

[134] The Guardian. The NSA files: *Decoded*. `https://www.theguardian.com/us-news/the-nsa-files`, November 2013.

[135] V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang. FPGA Implementation of Point Multiplication on Koblitz Curves using Kleinian Integers. In *Cryptographic Hardware and Embedded Systems*, CHES'06, pages 445–459, Berlin, Heidelberg, 2006. Springer-Verlag.

[136] J. van de Pol and N. P. Smart. Estimating Key Sizes for High Dimensional Lattice-Based Systems. In *IMA Int. Conf.*, volume 8308 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2013.

[137] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology — EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.

[138] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *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, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[139] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[140] V.S. Dimitrov, K.U. Järvinen, M.J. Jacobson, W.F. Chan, and Z. Huang,. Provably Sublinear Point Multiplication on Koblitz Curves and Its Hardware Implementation. In *Computers, IEEE Transactions on*, volume 57, pages 1469–1481, Nov. 2008.

[141] C. Vuillaume, K. Okeya, and T. Takagi. Defeating Simple Power Analysis on Koblitz Curves. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E89-A(5):1362–1369, May 2006.

[142] W. Wang and X. Huang. FPGA Implementation of a Large-Number Multiplier for Fully Homomorphic Encryption. In *IEEE International Symposium on Circuits and Systems (ISCAS 2013)*, pages 2589–2592, 2013.

[143] W. Wang and X. Huang. VLSI Design of a Large-Number Multiplier for Fully Homomorphic Encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):1879–1887, 2014.

[144] E. Wenger. Hardware Architectures for MSP430-Based Wireless Sensor Nodes Performing Elliptic Curve Cryptography. In *Applied Cryptography and Network Security — ACNS 2013*, volume 7954 of *Lecture Notes in Computer Science*, pages 290–306. Springer, 2013.

[145] E. Wenger and M. Hutter. A Hardware Processor Supporting Elliptic Curve Cryptography for Less than 9 kGEs. In *Smart Card Research and Advanced Applications — CARDIS 2011*, volume 7079 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2011.

[146] L. F. Williams, Jr. A Modification to the Half-interval Search (Binary Search) Method. In *Proceedings of the 14th Annual Southeast Regional Conference*, ACM-SE 14, pages 95–101, New York, NY, USA, 1976. ACM.

[147] Xilinx. *LogiCORE IP Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC wrapper (ug800)*, 3 2011. `http://www.xilinx.com/support/documentation/ip_documentation/ug800_v6_emac.pdf`.

[148] Xilinx. *Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC (ug368)*, 3 2011. `http://www.xilinx.com/support/documentation/user_guides/ug368.pdf`.

[149] Xilinx. *ML605 Hardware User Guide*, 2012. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

[150] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-Database Private Information Retrieval from Fully Homomorphic Encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, May 2013.

[151] ZDNet. The Internet of Things and Big Data: Unlocking the Power. http://www.zdnet.com/, March 2015.

Curriculum Vitae

Sujoy Sinha Roy was born on June 4th in Hooghly, India. He received the B.E degree in Electronics and Telecommunication Engineering from Bengal Engineering and Science University, Shibpur in 2007. He subsequently worked as an engineer in Tata Consultancy Services, Mumbai. In 2012, he received the M.S. degree in Computer Science and Engineering from Indian Institute of Technology, Kharagpur.

In September 2012, he started his PhD at the COSIC (Computer Security and Industrial Cryptography) research group at the Department of Electrical Engineering (ESAT) of the KU Leuven. His research area has been broadly in the field of efficient implementation of public key cryptography. His research has been generously funded by the European Commission through the Erasmus Mundus PhD Scholarship.

List of publications

Journals

1. Sujoy Sinha Roy, Frederik Vercauteren, Jo Vliegen, Ingrid Verbauwhede, "Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search", Accepted in IEEE Transactions on Computers as a regular paper. Preprint available on IEEE Xplore, DOI: 10.1109/TC.2017.2686385.

2. Sujoy Sinha Roy, Junfeng Fan, Ingrid Verbauwhede, "Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms", In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 5, pp. 810-818, May 2015. 2015.

3. Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray CC Cheung, Derek Pao, Ingrid Verbauwhede, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems", In IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 62, no. 1, pp. 157-166, Jan. 2015.

4. Oscar Reparaz, Sujoy Sinha Roy, Ruan de Clercq, Frederik Vercauteren, Ingrid Verbauwhede, "Masking ring-LWE", In Journal of Cryptographic Engineering, Springer Berlin Heidelberg, 2016, Vol.6(2), p.139-153.

5. Zhe Liu, Thomas Pöppelmann, Tobias Oder, Hwajeong Seo, Sujoy Sinha Roy, Tim Güneysu, Johann Großschädl, Howon Kim, Ingrid Verbauwhede, "High-Performance Ideal Lattice-Based Cryptography on 8-bit AVR Microcontrollers", Accepted in ACM Transactions on Embedded Computing Systems.

6. Kimmo Järvinen, Sujoy Sinha Roy, Ingrid Verbauwhede, "Arithmetic of τ-adic Expansions for Lightweight Koblitz Curve Cryptography", Under review in Journal of Cryptographic Engineering, Springer Berlin Heidelberg.

Conferences and workshops

1. Sujoy Sinha Roy, Angshuman Karmakar, Ingrid Verbauwhede, "Ring-LWE: Applications to Cryptography and Their Efficient Realization", In International Conference on Security, Privacy, and Applied Cryptography Engineering, Springer International Publishing, Volume 10076 of the book series Lecture Notes in Computer Science (LNCS).

2. Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, Ingrid Verbauwhede, "Additively homomorphic ring-LWE masking", In International Workshop on Post-Quantum Cryptography, Springer International Publishing, Volume 9606 of the book series Lecture Notes in Computer Science (LNCS).

3. Jeroen Bosmans, Sujoy Sinha Roy, Kimmo Järvinen, Ingrid Verbauwhede, "A Tiny Coprocessor for Elliptic Curve Cryptography over the 256-bit NIST Prime Field", In 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID).

4. Sujoy Sinha Roy, Kimmo Järvinen, Frederik Vercauteren, Vassil Dimitrov, Ingrid Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation", In International Workshop on Cryptographic Hardware and Embedded Systems CHES 2015, pp 164-184, Springer Berlin Heidelberg, Volume 9293 of the book series Lecture Notes in Computer Science (LNCS).

5. Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, Ingrid Verbauwhede, "Efficient Ring-LWE Encryption on 8-Bit AVR Processors", In International Workshop on Cryptographic Hardware and Embedded Systems CHES 2015, pp 663-682, Springer Berlin Heidelberg, Volume 9293 of the book series Lecture Notes in Computer Science (LNCS).

6. Sujoy Sinha Roy, Kimmo Järvinen, Ingrid Verbauwhede, "Lightweight coprocessor for Koblitz curves: 283-bit ECC including scalar conversion with only 4300 gates", In International Workshop on Cryptographic Hardware and Embedded Systems CHES 2015, pp 102-122, Springer Berlin Heidelberg, Volume 9293 of the book series Lecture Notes in Computer Science (LNCS).

7. Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, Ingrid Verbauwhede, "A masked ring-LWE implementation", In International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2015, pp 683-702, Volume 9293 of the book series Lecture Notes in Computer Science (LNCS).

8. Ruan De Clercq, Sujoy Sinha Roy, Frederik Vercauteren, Ingrid Verbauwhede, "Efficient software implementation of ring-LWE encryption", In DATE '15 Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition Pages 339-344.

9. Ingrid Verbauwhede, Josep Balasch, Sujoy Sinha Roy, Anthony Van Herrewege, "Circuit challenges from cryptography", In 2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers, San Francisco, CA, 2015, pp. 1-2.

10. Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, Ingrid Verbauwhede, "Compact ring- LWE cryptoprocessor", In International Workshop on Cryptographic Hardware and Embedded Systems CHES 2014: pp 371-391, Springer Berlin Heidelberg, Volume 8731 of the book series Lecture Notes in Computer Science (LNCS).

11. Sujoy Sinha Roy, Frederik Vercauteren, Ingrid Verbauwhede, "High precision discrete Gaussian sampling on FPGAs", In International Conference on Selected Areas in Cryptography, SAC 2013 pp 383-401, Springer Berlin Heidelberg, Volume 8282 of the book series Lecture Notes in Computer Science (LNCS).

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COMPUTER SECURITY AND INDUSTRIAL CRYPTOGRAPHY
Kasteelpark Arenberg 10, bus 2452
B-3001 Leuven
sujoy.sinharoy@esat.kuleuven.be
https://www.esat.kuleuven.be/cosic/