Secure Hardware Accelerators for Post Quantum Cryptography

Timo Zijlstra

UNIVERSITE
BRETAGNE
LOIRE / MATHSTIC

Université Bretagne Sud
ubs

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE Nº 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Timo ZIJLSTRA »

« Accélérateurs matériels sécurisés pour la cryptographie post-quantique »

Thèse présentée et soutenue à « Lorient », le « 28/09/2020 »
Unité de recherche : Lab-STICC
Thèse Nº : 564

Rapporteurs avant soutenance :

Lilian BOSSUET	Professeur des Universités, Université Jean Monnet Saint-Étienne
Régis LEVEUGLE	Professeur des Universités, Grenoble INP, Université Grenoble Alpes

Composition du Jury :

Président :	Jean-Claude BAJARD	Professeur des Universités, Université de Paris (UPMC)
Examinateurs :	Nele MENTENS	Professeur des Universités, Université Leiden / Université KU Leuven
	Jean-Claude BAJARD	Professeur des Universités, Université de Paris (UPMC)
Dir. de thèse :	Arnaud TISSERAND	Directeur de Recherche, CNRS
Co-enc. de thèse :	Karim BIGOU	Maître de conférences, Université de Bretagne Occidentale

Invité(s) :

Benoît GÉRARD	Expert, DGA

Contents

Chapter 1

Introduction

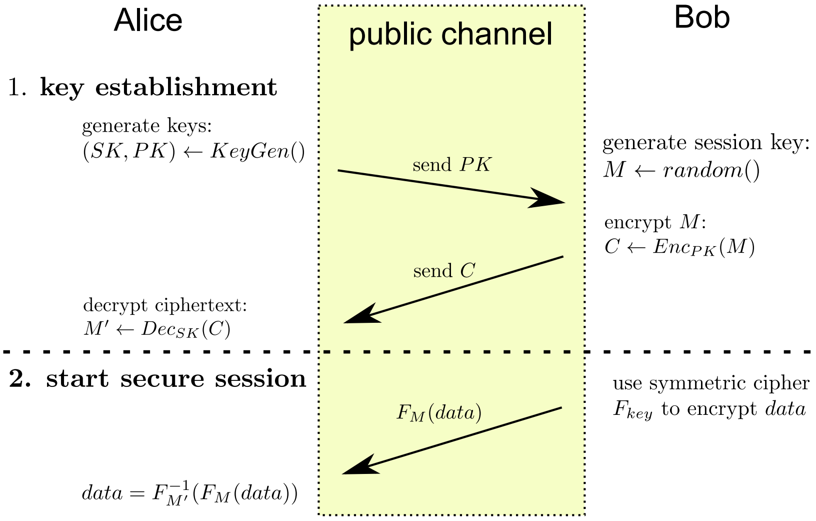
1.1 Context

An increasing number of electronic devices require the ability to exchange data in a secure manner. A secure application for sending and receiving messages for example, must ensure the confidentiality and authenticity of exchanged messages. *Encryption* is the cryptographic tool that is used to guarantee confidentiality. It uses mathematical functions that are easy to compute but hard to invert. Only with a so-called *key* the inverse function can be computed. An encryption of the message is obtained by applying such a function to the message, or *plaintext.* The content of the encrypted message, or *ciphertext*, can only be read by those who hold the key to invert the encryption function. Inverting the encryption and recovering the message is called *decryption.* Symmetric-key cryptography deals with the encryption and decryption of the message, assuming that both sender and receiver have the key to encrypt and decrypt. It is symmetric in the sense that the same key must be used for both encryption and decryption.

Symmetric-key encryption requires that both sender and receiver already have a shared key. Before being able to use symmetric-key encryption, such a key must be created and shared in a secure way. A key exchange protocol consists of an (unsecure) exchange of data over a public channel between two parties Alice and Bob. The goal of the protocol is to have both parties ending up with the same piece of pseudo-random information, which can then be used as a key for use in symmetric-key encryption. It must be ensured that the public data exchanged over the public channel do not reveil any information about the shared key created using this data.

The data exchanged between Bob and Alice is sent over a public channel (see figure 1.1). It is assumed that there is always an eavesdropper that is able to intercept anything sent over this channel. This makes a secure key establishment particularly difficult. The problem is that any straightforward method of sharing a key would give away information about the key to the eavesdropper. The solution is found in a clever use of *computationally hard problems.* These are mathematical problems for which no efficient algorithm is known. Such a problem cannot be solved within a reasonable computation

Figure 1.1: Using symmetric and asymmetric cryptography to exchange confidential data from Bob to Alice. SK is the secret key, PK the public key and C the ciphertext. If C is decrypted correctly, then $M = M'$ and the *data* that Alice obtains is equal to Bob's *data*.



time. Computationally hard problems allow to construct *trapdoor* functions, which are typically used as decryption function in public-key encryption schemes.

The discrete logarithm is an example of a computationally hard problem. Let g be a multiplicative generator of $(\mathbb{Z}/p\mathbb{Z})^*$ for some prime number p. The discrete logarithm problem is to find the value of a given g and $g^a \bmod p$ for some integer a. The number of computations needed to solve this problem is proportional to the size of the parameter p. To find the discrete logarithm of $g^a \bmod p$ using a brute force approach, one must compute $g^i \bmod p$ for increasing i until finding a value for which $g^i \equiv g^a \bmod p$. For sufficiently large p, it may take years to solve an instance of the discrete logarithm problem. Key exchange protocols are designed in such a way that the (secret) key can only be deduced from the (publicly) exchanged data by solving the underlying hard problem.

The Diffie-Hellman protocol [47] uses a prime number p and a multiplicative generator g of $(\mathbb{Z}/p\mathbb{Z})^*$ as public parameters. Alice and Bob pick a random integer a and b and compute the values $g^a \bmod p$ and $g^b \bmod p$ respectively. These values are then sent over the public channel. The random integers a and b are kept secret. Alice receives $g^b \bmod p$ and computes $(g^b)^a \bmod p$. Bob, who picked b, receives $g^a \bmod p$ and computes $(g^a)^b \bmod p$. Since $(g^a)^b \equiv g^{ab} \equiv (g^b)^a \bmod p$, both Alice and Bob end up with the same element $g^{ab} \bmod p$. This element forms the secret key to be used with a symmetric encryption method for the remainder of the encrypted communication between Alice and Bob.

The eavesdropper on the public channel intercepts $g^a \bmod p$ and $g^b \bmod p$. To compute $g^{ab} \bmod p$, either a or b has to be computed using only the information $g^a \bmod p$ and $g^b \bmod p$. This is the discrete logarithm problem described above: to *break* the protocol (i.e. to find the key), one must compute a discrete logarithm. Assuming that p is sufficiently large, it is not practical to solve this problem and therefore the key exchange is secure. In order for the protocol to be useful in practice, it is essential that the number of computations needed by the eavesdropper to break the scheme, be significantly greater than the number of computations performed by the two legitimate parties. In the worst case, the eavesdropper will compute $g^i \bmod p$ for integer $0 < i < p-1$, before finding a or b. Alice and Bob on the other hand, only perform two exponentiations each. Let $n = 1 + \lfloor \log_2(p) \rfloor$, that is, n is equal to the length of the binary representation of p. The worst-case hardness of breaking the Diffie-Hellman protocol is exponential in n: between 2^{n-1} and 2^n exponentiations have to be computed. Increasing by 1 the size of p doubles the hardness of breaking the protocol. A Diffie-Hellman instance of parameters g and p where $1 + \lfloor \log(p) \rfloor$ is equal to $n+1$ (instead of n), is twice as hard to break. The parameter n is used as a *security parameter*. The U.S. National Institute of Standards and Technology (NIST) recommends using $n = 2048$ or even $n = 3072$ for higher security levels [15].

On average, the brute force algorithm solves the discrete logarithm in $\Theta(2^n)$ exponentiations. There exist various algorithms that compute discrete logarithms faster than the brute force approach, significantly lowering the security of the protocol. To compensate, the parameter n (and therefore also p) has to be increased. A side effect of this increase is that Alice and Bob have to perform more operations and on larger values to compute their 2 exponentations, and the size of $g^a \bmod p$ and $g^b \bmod p$, sent over the public channel, is increased. Increasing the parameter n too much, may make the protocol impractical for use because of computation time requirements or bandwidth limitations. Luckily, the fastest discrete logarithm algorithms are of a complexity that is at least subexponential in the parameter n. This means that a linear increase of n causes a subexponential increase in the computation time of the discrete logarithm algorithms. The modification to n required to make the computation of these algorithms impractical, is acceptable in terms of computation time and bandwith requirements for the Alice and Bob.

The Diffie-Hellman protocol is not limited to the group $(\mathbb{Z}/p\mathbb{Z})^*$. Any cyclic group $G = \langle g \rangle$ in which the discrete logarithm is significantly harder to compute than its group operation, can be used. Another example is the additive group of points generated by some point P on an elliptic curve over some finite field $\mathbb{K}$. The *discrete logarithm* in such a group consists of finding some scalar $a \in \mathbb{K}$ given the generator P and the point aP on the curve. For the right choice of parameters, this is a hard problem. More background on elliptic curve cryptography (ECC) can be found in [39].

Public-key encryption is a slightly different approach to creating and sharing a symmetric encryption key. It allows Alice and Bob to exchange a small message without already having a pre-shared symmetric key. In practice, this small message consists of a symmetric encryption key, therefore it does not differ from key exchange protocols in its main purpose. The RSA public-key encryption scheme [104] is the most famous example. The security of RSA is not based on the hardness of the discrete logarithm problem, but instead on the hardness of the integer factorization problem. This problem consists of factorizing an integer that is a product of two large prime numbers. Given a number $N = p \cdot q$ for two large prime numbers p and q, it is very hard to find the factors p and q. As is the case for the Diffie Hellman protocol, the security of RSA depends on the complexity of the algorithms solving the underlying problem of integer factorization.

The integer factorization problem and the discrete logarithms over certain groups can be solved in polynomial time by Shor's quantum algorithm [108]. This means that, using a quantum computer, these problems are easy to solve. The cryptographic protocols based on the hardness of these problems, such as RSA, Diffie Hellman and ECC, are broken against attackers using sufficiently large quantum computers. While the quantum computers that are operational are still too small to pose a threat today, development is in progress. It is difficult to predict exactly when there will be quantum computers that are sufficiently large to to break RSA and ECC. Some experts believe that it might happen in the next 20 years [89], with a $\frac{1}{2}$ chance of breaking RSA-2048 by 2031. Development in practical quantum computing depends on the ability to produce reliable fault-resistant qubits and its scalability. Regardless of the accuracy of this prediction, there are various reasons to start developing quantum-safe alternatives to RSA/ECC. It may take a lot of time to develop and standardize new cryptographic algorithms. The NIST hash function competition, with its goal to standardize a secure hashing algorithm, lasted 5 years. It should be assumed that encrypted communication is stored by adversaries that may in the future use quantum computers to break the encryption. Therefore it is not sufficient to use cryptographic algorithms that we know will become obsolete. Some communication is supposed to remain secret for over 20 years. For these cases it is imperative to use alternatives to RSA/ECC that will be resistant against quantum algorithms.

These alternatives are called *post-quantum cryptography* (PQC). The NIST competition for the standardization of post-quantum cryptography was launched in 2016 [36]. Its goal is to select and standardize post-quantum algorithms for public-key encryption and digital signatures. The standardized algorithms should in time replace RSA/ECC. The security of post-quantum primitives relies on mathematical hard problems for which quantum computers do not offer significant speedup, as opposed to the large integer factorization problem. The most studied proposed post-quantum algorithms fall in one of four different catagories:

- Lattice-based cryptography uses hard problems like Learning with Errors (LWE) [100] and NTRU [59], which are both related to the Shortest Vector Problem (SVP). These problems have been studied intensively, and known quantum algorithms cannot solve them efficiently. Schemes based on LWE and NTRU use relatively simple algebraic structures, such as matrices, vectors and polynomials. They are known to be fast to compute, especially the variants using polynomial ring structures. An important drawback to lattice-based cryptography is the size of the keys and ciphertexts when compared to ECC. The size of the public key of the Frodo cryptosystem for example, is 21.6 kB (see Table 1.1), while the total communication of the ECC Diffie-Hellman key exchange does not exceed 1 kB [16]. Other examples include Kyber [27], NewHope [4] and NTRUPrime [21].

- Code-based cryptography is based on the problem of decoding a random linear code. Code-based cryptosystems use some error correcting code $\mathcal{C}$ that can be decoded using a fast decoding algorithm $\mathcal{A}$. The specification of the code, in the form of a matrix, is kept as part of the secret key. The public key is created by applying linear transformations and permutations to the secret matrix in order to obtain a scrambled code. This scrambled code is the public key and can be used by anyone to encode messages. The ciphertext then consists of a codeword in the scrambled code. The holder of the secret key can undo the linear transformations and permutations to the codeword and use her fast decoding algorithm $\mathcal{A}$ to decode and obtain the plaintext. The McEliece cryptosystem [85] using Goppa codes was proposed in 1978 and has never been broken. It is not much used in practice, due to the large public key (over 1 MB, see Table 1.1) which makes it impractical. Variants of the McEliece scheme use different techniques in order to decrease the key size, such as replacing the Goppa code by some other linear code.

- Isogeny-based cryptography uses the Diffie-Hellman key exchange framework. Instead of the group $(\mathbb{Z}/p\mathbb{Z})^\times$ or the additive group of points on elliptive curves, a different group is used. An isogeny graph is defined by vertices representing elliptic curves and edges between each pair of elliptic curves that is connected by some isogeny. The group operation enables random walks over such an isogeny graph. It is assumed that it is computationally hard to solve the problem of finding an explicit isogeny between two given elliptic curves. It also seems that quantum computers offer no significant speed-up in solving this problem. Advantages of isogeny-based cryptography include small key sizes (less than 1 kB, also see Table 1.1) and the ressemblance to the existing Diffie Hellman key exchange. An important inconvenient is the long computation time needed to compute the group action. SIKE [10] is the only isogeny-based submission in the NIST post-quantum contest.

- Multivariate polynomial cryptography is based on the hardness of solving a system of multivariate quadratic equations. It seems most promising for digital signatures but is not used much for public-key encryption.

The submissions to the NIST post-quantum contest differ in many aspects, and it can be challenging to make a fair comparison. The theoretical security is one of the criteria that must be satisfied. The submissions focus on several pre-defined security levels. Their claims that certain security levels are obtained by their proposal, are to be verified by cryptanalysts. The practicality of proposed solutions is a second criterium. In real world applications, there are constraints on computation time and transmission of data. The size of the public keys and ciphertexts is an important measure of the efficiency of a cryptosystem. A key establishment mechanism typically requires the transmission of one public key and one ciphertext. The sum of the public key and ciphertext size is equal to the total

[1] Key and ciphertext sizes and cycle count are for the parameter set for a security equivalent to AES-192.

[2] Key and ciphertext sizes and cycle count are for the parameter set targeting only 128 bits of security.

Table 1.1: A selection of round 2 submissions to NIST's post-quantum PKE contest. Key and ciphertext sizes are for parameter sets of the highest security level (equivalent to AES-256) unless stated otherwise. The cycle count for the computation of the encryption algorithm is taken from [22] who used an Intel Core i3-2310M processor.

Cryptosystem	category	type	public key (kB)	secret key (kB)	ciphertext (kB)	cycles ($\times 10^3$)
Classic McEliece [20]	code	binary Goppa	1,044	13.9	0.2	318
BIKE-1 [6]	code	QC-MDPC	8.2	0.5	8.2	802
Frodo [28]	lattice	LWE	21.6	21.6	21.6	52,672
NewHope [4]	lattice	RLWE	1.8	1.8	2.2	498
Kyber [27]	lattice	MLWE	1.6	1.6	1.6	471
Saber [45]	lattice	MLWR	1.3	1.8	1.5	377
NTRUPrime [21] [1]	lattice	NTRU	1.2	0.6	1.0	11,032
SIKE [10] [2]	isogeny	supersingular	0.4	0.1	0.4	26,587

transmission required for one key establishment. Table 1.1 shows key and ciphertext sizes for a number of candidates in the NIST's post-quantum contest. Besides the size of the data transmission, speed is another essential measure. A solution that is very secure but requires a long computation time, may not be practical. The cycle counts shown in Table 1.1 are very high for some schemes. Improvement to satisfy computation time constraints can be obtained using hardware acceleration. Hardware units dedicated to the computation of specific cryptographic algorithms are faster than general purpose processors. The efficiency of hardware implementations is therefore an indicator of the practicality of a proposed cryptographic algorithm.

The security of hardware implementations does not only depend on the hardness of the mathematical problem underlying the cryptosystem, but also on the way it is implemented. Side-channel analysis (SCA) [71] makes use of physical properties that are measured while the device under attack is running. These physical properties often depend on the exact values that are being used in the computations. During decryption, the secret key is part of these values. Measuring the exact time of the computation of $g^a \bmod q$ during a Diffie-Hellman key exchange for example, may reveal information about the secret value a. In fact, if the exponentation is implemented using the Square-and-Multiply algorithm [86], the computation time depends directly on the binary representation of a. This algorithm iterates over the zeroes and ones in the binary representation of a and computes a square operation if a bit is equal to 0, while computing both a square and a multiplication if a bit is equal to 1. If the exponentation takes relatively little time to compute, one can deduce that the binary representaion of a contains many zeroes. On the other hand, if the exponentation takes a long time to compute, there must be many ones in the binary representation of a. The Diffie-Hellman key exchange can thus be attacked, without attacking the underlying mathematical problem. These attacks are called *side-channel attacks*. The computation time is referred to as a *side channel*.

Another example of a side channel is the power consumption of a device executing a cryptographic operation. Power analysis uses measured power *traces*, showing the exact power consumption of a device during the computation of an algorithm that uses the secret key. In the example of the Square-and-Multiply algorithm, power analyis may reveal the complete secret key a. During a multiplication a device consumes more power than during a square. The power trace thus reveals a unique pattern of squares and multiplications that correspond to zeroes and ones of the secret key. The same attack applies to the exponentation during an RSA decryption, and even to scalar multiplication in ECC using the Double-and-Add algorithm [58] to compute a (secret) multiple of a point on the curve. These attacks use only one power trace and are therefore called *simple* power attacks (SPA).

Using more traces, less straightforward relations between power consumption and secret keys may be detected. It is often assumed that there is a linear relation between the power consumption and the number of non-zero bits in the registers at any given moment during a computation. This is called the *Hamming Weight model* [82]. The power consumption of a device can thus be *modelled* for given inputs. Another model is defined by the *Hamming Difference*, in which it is assumed that the power consumption depends linearly on the number of bits that change value in the registers at any given moment. It is common practice to focus on a small part of the (unknown) secret key and the exact period that this part is being manipulated by the algorithm. The power consumptions for all the possibilities of the small part of the secret key are predicted by the attacker, using a prediction model. Then these predictions are compared to the observed power trace. Statistical tools are used to determine which one of the predictions fits the observed pattern most accurately. Correlation Power Analysis [30][84] observes and predicts many traces for many known inputs, and uses the Pearson's correlation coefficients to select the most likely correct value for the small part of the secret key. The attack then continues for the next small part of the secret key, until the full key is recovered. Altenatively, the side-channel attacker may stop at the point at which a large enough part of the secret key is known, so that a brute force approach can be used to recover the remaining unknown parts of the key.

Side-channel attacks show that the theoretical security of a cryptographic protocol is not sufficient to guarantee the security of its implementation in practice. Implementations should be protected by countermeasures against side-channel attacks. Timing attacks, as the one described on the Square-and-Multiply algorithm, can be avoided by making the implementation *constant time*. The computation time of constant-time implementations does not depend directly or indirectly on the secrets in the algorithm. A classic method of making the Square-and-Multiply algorithm constant time, is to add dummy operations. Whenever a zero appears in the binary representation of the secret key, not only a square is computed, but also a multiplication of some dummy variable. This ensures that the final result does not change, while timing attacks are avoided by constantly squaring and multiplying, independently of the secret key. This countermeasure may also prevent SPA, provided that the other potential sources of information leakage, such as the control part of the implementation, are protected as well.

Power attacks using multiple traces may be avoided by *randomizing* the computations. If the operations in the algorithm are linear, then *masking* may be used. This method involves generating a random value s' and adding it the the secret key s. The algorithm is then computed twice: the first time using s' as secret key, and the second time using $s + s'$. The two results are then recombined to obtain the correct outcome. The algorithm computes on randomized values only, provided that during each call to the algorithm a new random value s' is generated. The randomized computations yield randomized power traces for the attacker observing the power consumption of the device. There is no correlation between the random traces and those predicted (using the Hamming Weight model). Masking can thus thwart attacks using correlation power analysis. A major drawback of masking is that the algorithm is essentially computed twice by the device, dramatically increasing computation time. Moreover, the presence of non-linear operations in the cryptographic algorithm might make the masking method impractical. Other randomization methods may be possible, depending on the cryptographic algorithm to be protected.

Among physical attacks, *fault injection* constitute another common threat for cryptographic implementations, but this aspect is not addressed in this thesis due to lack of time and kept for future prospects.

1.2 Objective and outline of the thesis

In this thesis we will study hardware acceleration for various lattice-based encryption schemes in PQC. We implement LWE, Ring-LWE (RLWE) and Module-LWE (MLWE) based public-key encryption schemes on FPGA in order to compare their performances in hardware for multiple parameter sets, degrees of parallelism and other implementation and algorithm choices. The aim is to obtain implementations that are fast but use limited resources. In this thesis, we take a particular interest in hardware security. We analyse side-channel vulnerabilities and propose countermeasures against side-channel attacks. The countermeasures are implemented on FPGA to evaluate their cost in terms of area and computation time. Various trade-offs between security and computation time provide insight in the practicality of the post-quantum cryptosystems.

In chapter 2 we provide definitions and notations commonly used in the domain. This chapter briefly provides some mathematical notions required in order to read this thesis. In lattice cryptography, a basic understanding of finite fields and linear algebra is needed in order to appreciate the functionality of the cryptosystems.

Chapter 3 provides further background and the state of the art of lattice based cryptography. Mathematical problems involving lattices, such as the shortest vector problem, are introduced. Cryptographic constructions rely on the computational hardness of solving these problems. We discuss various algorithms to compute the operations required by the cryptosystems, such as modular reduction and polynomial multiplication. An overview of the state of the art of side-channel attacks, with a focus on lattice cryptography, is presented in section 3.6.

In chapter 4 we discuss the methodology used for our FPGA implementations. Our method relies on High Level Synthesis (HLS), a tool that can be used to generate FPGA implementations starting from a description in a high level programming language, such as C or C++. Section 4.1 provides an introduction to FPGAs and HLS and discusses the application of HLS in cryptography. The use of HLS is not very common among cryptographers, who typically use VHDL/Verilog languages for FPGA implementations. The HLS synthesis of the modular reduction operator in C (%), essential in public-key cryptography, yields poor performances. Section 4.2 aims to improve the computation of modular reduction by replacing the % operator. We implement finite-field arithmetic on FPGA using HLS and compare the implementation results for different modular reduction algorithms, implementation styles and parameters. In section 4.3 we explore the implementation of polynomial multiplication, an important component of lattice based cryptosystems. We compare implementation results of the schoolbook algorithm using different coding styles. We evaluate the impact of parallelization on the speed and area utilization of the implementations.

In chapter 5 we implement lattice based cryptosystems on FPGA. The objective of this chapter is to provide a comparison of the practicality of the implemented cryptosystems. The differences between the algorithms used by LWE, RLWE and MLWE based public-key encryption make some cryptosystems more practical than others. LWE based cryptosystems typically suffer from low performance compared to RLWE or MLWE. The bottleneck in the LWE encryption algorithm is the multiplication of large matrices. To accelerate this multiplication, we study the effectiveness of parallelism in the FPGA implementation. By dividing the computations over a number of dedicated hardware units, the computation time can be reduced. The same effort is made for the implementations of RLWE and MLWE based encryption, where polynomial multiplication is the most important arithmetic operation. This chapter also discusses randomness generation and chosen ciphertext attack (CCA) secure implementations. All lattice based public-key algorithms rely on the ability to generate random bits. Efficient pseudo-random number generators (PRNG) are therefore an important part of the implementation. Fast PRNGs however, are less cryptographically secure than the more conservative ones. This results in another trade-off between security and performance, discussed in section 5.6.

The topic of chapter 6 is the hardware security of our RLWE based decryption implementation. We highlight the vulnerabilities of the algorithm and show how these vulnerabilities might be exploited by an attacker using side-channel attacks. To prevent side-channel attacks, various countermeasures are proposed. Our countermeasures are based on the randomization of the operations, in order to randomize the information leakage through side channels. State of the art countermeasures include masking, shifting and blinding. We discuss these protections and present various improvements. We propose new countermeasures, such as a redundant representation of finite-field elements, and two shuffling techniques. All of the countermeasures from the state of the art, with improvements, and our proposed countermeasures are implemented on FPGA. We compare the difference in computation time and area utilization for each countermeasure implementation. In section 6.6, the countermeasures are generalized in order to be applied to LWE and MLWE based implementations.

Chapter 2

Definitions and Notations

- $\mathbb{Z}$ is the ring of integers. The $\mathbb{Z}$-module $\mathbb{Z}^n$ consists of vectors of length n whose coefficients are in $\mathbb{Z}$. Vectors are written in bold font: $\mathbf{a} \in \mathbb{Z}^n$.

- $\mathbb{R}$ is the field of real numbers.

- For any $a \in \mathbb{R}$, $\lfloor a \rfloor$ is the integer $a' \in \mathbb{Z}$ that is closest to a and for which $a' \leq a$. The integer a'' closest to a such that $a'' \geq a$ is noted $\lceil a \rceil$.

- For any integer $q > 0$, $\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}$ is the ring of integers modulo q, represented by the q elements $\{0, 1, \ldots, q-1\}$. In this ring two elements are equivalent if and only if their difference in $\mathbb{Z}$ is an integer multiple of q. Equivalence is noted with the $\equiv$ symbol, for instance $q \equiv 0 \bmod q$. If q is a prime number, then $\mathbb{Z}_q$ is a field, that is, each non-zero element a in $\mathbb{Z}_q$ has an inverse a^{-1} such that $a^{-1}a \equiv 1 \bmod q$. In $\mathbb{Z}_5$ for example: $2 \cdot 3 = 6 \equiv 1 \bmod 5$, therefore the inverse 2^{-1} of 2 in $\mathbb{Z}_5$ is equal to 3. The inverse of 3 in $\mathbb{Z}_5$ is 2 and $4^{-1} \bmod 5 = 4$ because $4 \cdot 4 = 16 \equiv 1 \bmod 5$.

  The multiplicative group of inversible elements of $\mathbb{Z}_q$ is noted $\mathbb{Z}_q^\times$. Each element a in this group has a multiplicative order $Ord(a)$. The order of some a is the smallest non-zero integer i for which $a^i \equiv 1 \bmod q$. A primitive n-th root of unity $\omega \in \mathbb{Z}_q$ is an element of order n. The order of each element in $\mathbb{Z}_q^\times$ divides the number of elements in $\mathbb{Z}_q^\times$. If q is prime, then $\mathbb{Z}_q^\times = \mathbb{Z}_q \backslash \{0\}$ and there exists an element $g \in \mathbb{Z}_q^\times$ such that $Ord(g) = q - 1$. Such an element is called a *generator* of $\mathbb{Z}_q^\times$, since it generates the group: $\mathbb{Z}_q^\times = \{g, g^2, g^3, \ldots, g^{Ord(g)} = 1\}$.

- $\mathbb{Z}[x]$ is the ring of polynomials with coefficients in $\mathbb{Z}$. Its contains all the elements in the set:

  $$S = \Big\{ \sum_i a_i x^i : a_i \in \mathbb{Z} \text{ for } i = 0, 1, 2, \ldots \Big\}$$

  Polynomials are written in bold font: $\mathbf{a} \in \mathbb{Z}[x]$. Addition in $\mathbb{Z}[x]$ is defined by the map

$$
\begin{aligned}
+ : S \times S &\longrightarrow S \\
\left(\sum_i a_i x^i, \sum_j b_j x^j \right) &\longmapsto \sum_i (a_i + b_i) x^i
\end{aligned}
$$

that is, the sum of two polynomials is obtained by computing the sum of each pair of coefficients (a_i, b_i) for each degree i. Multiplication of two elements in $\mathbb{Z}[x]$ is computed by the map:

$$
\begin{aligned}
\bullet : S \times S &\longrightarrow S \\
\left(\sum_i a_i x^i, \sum_j b_j x^j \right) &\longmapsto \sum_i \sum_{j \leq i} a_j b_{i-j} x^i
\end{aligned}
$$

Let $\mathbf{a} \in \mathbb{Z}[x]$. The *ideal* generated by $\mathbf{a}$ is noted $(\mathbf{a})$. It contains all elements in the set

$$
(\mathbf{a}) = \{\mathbf{ab} : \mathbf{b} \in \mathbb{Z}[x]\} .
$$

If for instance $\mathbf{a} = x^2$, then $(\mathbf{a}) \subset \mathbb{Z}[x]$ is the set of polynomials whose first and second coefficients are equal to zero: $(\mathbf{a}) = \left\{ \sum_{i \geq 2} a_i x^i : a_i \in \mathbb{Z} \text{ for } i = 2, 3, \ldots \right\}$. The ring $\mathbb{Z}[x]/(\mathbf{a})$ is defined by the ring of polynomials in which two elements are considered to be equivalent if and only if their difference is in $(\mathbf{a})$. Let $\mathbf{a} = x^2$ for example, so that $\mathbb{Z}[x]/(\mathbf{a}) = \mathbb{Z}[x]/(x^2)$. Elements $x + 3$ and $x^7 + x^4 + x + 3$ are equivalent since their difference is $x^7 + x^4 = x^2(x^5 + x^2) \in (x^2)$, therefore $x + 3 \equiv x^7 + x^4 + x + 3$ in $\mathbb{Z}[x]/(x^2)$. Each polynomial is equivalent to some polynomial of degree smaller than 2. The ring $\mathbb{Z}[x]/(x^2)$ can therefore be represented by the elements of the set $\{a_0 + a_1 x : a_0, a_1 \in \mathbb{Z}\}$.

- $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ for some integer n, is the ring of polynomials in which two elements are equivalent if and only if their difference in $\mathbb{Z}[x]$ is a multiple of $x^n + 1$. In other words: $x^n + 1 \equiv 0$. Polynomials in this ring are of degree smaller than n, because $x^n \equiv -1$.
- $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. This is a finite ring, i.e. there is a finite number of elements in this ring. This number is equal to q^n, which is the number of polynomials of degree $< n$ with coefficients in $\mathbb{Z}_q$. If there exists a $2n$-th primitive root of unity ϕ in $\mathbb{Z}_q$, then $\phi^n \equiv -1 \bmod q$ and the map

$$
\begin{aligned}
\sigma : \mathcal{R}_q &\longrightarrow \mathbb{Z}_q^n \\
\mathbf{a}(x) &\longmapsto \big(\mathbf{a}(\phi), \mathbf{a}(\phi^3), \mathbf{a}(\phi^5), \dots, \mathbf{a}(\phi^{2n-1})\big)
\end{aligned}
$$

defines an isomorphism between $\mathcal{R}_q$ and $\mathbb{Z}_q^n$.

- An *error distribution* χ is a symmetric probability distribution centered around 0. The error distribution χ over the integers is denoted $\chi(\mathbb{Z})$. Samples from error distributions must be close to 0 with high probability. Examples include the Gaussian $\mathcal{N}(0, \sigma)$ distribution and the distribution $\mathcal{U}([-a, a])$ that is uniform on the interval $[-a, a]$ for some a close to zero.
- $\mathcal{B}_\lambda$ is the binomial (n, p) distribution with $p = \frac{1}{2}$ and integer parameter $n = \lambda$ centered around 0.

List of acronyms

AES	Advanced Encryption Standard
BRAM	Block RAM
CPA	Chosen Plaintext Attack
CC	Clock Cycles
CCA	Chosen Ciphertext Attack
CVP	Closest Vector Problem
DIF	Decimation-In-Frequency
DIT	Decimiation-In-Time
DPA	Differential Power Analysis
DSP	Digital Signal Processing block
ECC	Elliptic Curve Cryptography
FPGA	Field Programmable Gate Array
$\textsc{GapSVP}_\gamma$	Decisional Approximate Shortest Vector Problem
HLS	High Level Synthesis
HW	Hamming Weight
KEM	Key Exchange Mechanism
LFSR	Linear Feedback Shift Register
LUT	Look up Table
LWE	Learning With Errors
LWR	Learning With Rounding
LSB	Least Significant Bit

MLWE	Module Learning With Errors
MLWR	Module Learning With Rounding
MSB	Most Significant Bit
NIST	National Institute of Standards and Technology (U.S.)
NTT	Number Theoretic Transform
PCC	Pearson's Correlation Coefficient
PKC	Public Key Cryptography
PKE	Public Key Encryption
PRNG	Pseudo Random Number Generator
RLWE	Ring Learning With Errors
RLWR	Ring Learning With Rounding
RSA	Rivest Shamir Adleman (cryptosystem)
RNG	Random Number Generator
SCA	Side Channel Analysis
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm with Keccak
SPA	Simple Power Analysis
SVP	Shortest Vector Problem
SVP_γ	Approximate Shortest Vector Problem
TRNG	True Random Number Generator

Chapter 3

State of the Art

3.1 Introduction

This chapter introduces cryptographic primitives, computationally hard lattice problems and the concept of implementation security. Section 3.2 explains the mechanisms used in public-key encryption, with RSA as an example. Lattice based PKEs rely on the hardness of lattice problems. A number of these mathematical problems are discussed in section 3.3. This section also provides a description of the first LWE based cryptosystem. Variations, optimizations and generalizations of this cryptosystem are discussed in section 3.4. Section 3.5 deals with the computational aspect of the presented schemes. This includes algorithms for modular reduction and polynomial multiplication in finite rings. Finally, section 3.6 provides the state of the art of side-channel attacks. Different methods, ranging from simple-power analysis to fault attacks, are introduced. The most relevant side-channel attacks on lattice based cryptography from the state of the art are presented. Countermeasures to improve the security of implementations against these attacks are also discussed.

3.2 Public-Key Encryption

A public-key encryption scheme, or *cryptosystem*, is defined by three publicly known algorithms: key generation ($\mathcal{G}$), encryption ($\mathcal{E}$) and decryption ($\mathcal{D}$). Algorithm $\mathcal{G}$ generates a secret key SK and corresponding public key PK. The encryption takes as input a public key and a message (*plaintext*) and returns the ciphertext that encrypts the message using the public key. The decryption takes a ciphertext and a secret key and computes the plaintext. The algorithms must satisfy the following properties [47]:

1. For any plaintext m,

$$\mathcal{D}(SK, \mathcal{E}(PK, m)) = m,$$

that is, the decryption of an encrypted message is equal to the plaintext.

2. It is easy to evaluate the algorithms $\mathcal{E}$ and $\mathcal{D}$ for any input.

3. Given only the public key PK and a ciphertext $c = \mathcal{E}(PK, m)$, there is no easy way to compute m.

4. For all possible secret keys SK, algorithm $\mathcal{G}$ can compute a public key PK.

RSA [104] and the McEliece cryptosystem [85] were the first examples of public-key encryption schemes. To generate a pair of RSA keys, two large prime numbers p and q are needed, and kept secret. The public key is given by $M = p \cdot q$ and some random number e that is inversible mod $(p-1)(q-1)$. The secret key consists of $d = e^{-1}$ mod $(p-1)(q-1)$.

Algorithm 1 RSA encryption and decryption

```
Encryption of plaintext $\mu \in \mathbb{Z}_M$ using public key $PK = (M, e)$.
function Enc($\mu, PK$)
    return $\mu^e \bmod M$
Decryption of ciphertext $c \in Z_m$ using secret key $SK = d$.
function Dec($c, SK$)
    return $c^d \bmod M$
```

The decryption of a ciphertext returns the plaintext. Using Fermat's Little Theorem, it holds that:

$$\begin{aligned} c^d &= \mu^{ed} \bmod M \\ &= \mu^{1+k(p-1)(q-1)} \bmod M, \text{ for some } k \in \mathbb{Z} \\ &= \mu \cdot (1 \bmod M)^k \\ &= \mu. \end{aligned}$$

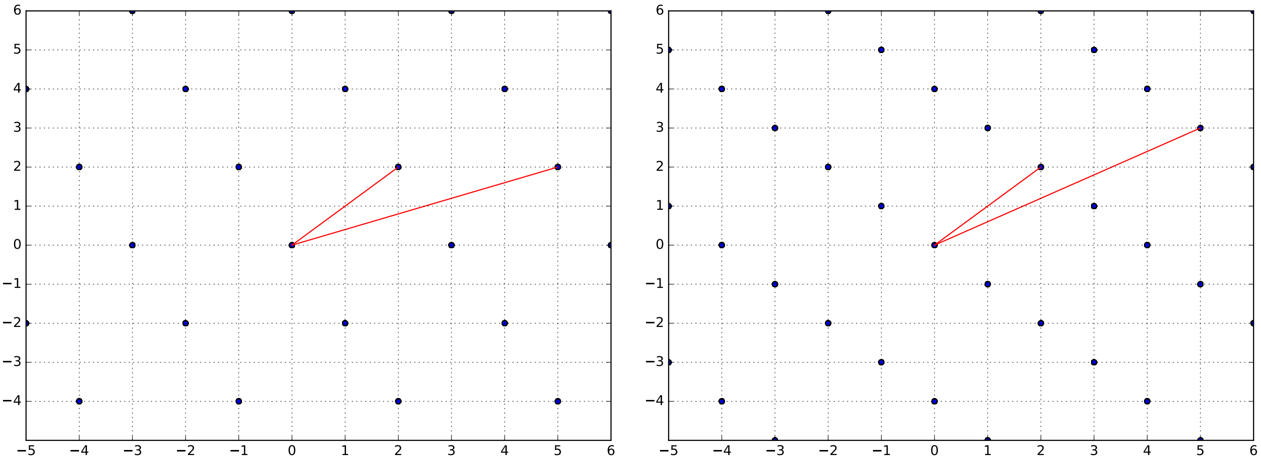
The security of RSA relies on the hardness of the factorization problem. An attacker who factors the public key $M = p \cdot q$, can compute the inverse of e mod $(p-1)(q-1)$.

3.3 Lattice Problems

A lattice L is a subgroup of $\mathbb{R}^n$ defined by a set of basis vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. It contains all linear integer combinations of the basis vectors [100]:

$$L = \Big\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z} \Big\}$$

Figure 3.1: Examples of lattices of dimension 2 over $\mathbb{Z}/12\mathbb{Z}$. The one on the left is generated by basis vectors $(2, 2)$ and $(5, 2)$, the one on the right is generated by $(2, 2)$ and $(5, 3)$.

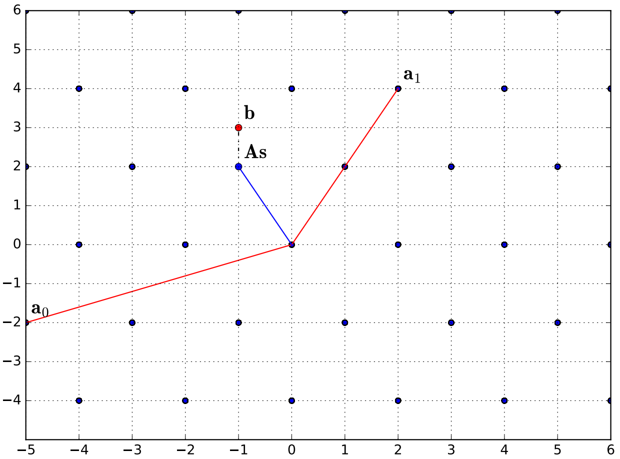


Examples of lattices of dimension 2 over the finite ring $\mathbb{Z}/12\mathbb{Z}$ are shown in Figure 3.1. The lattice generated by $\mathbf{b}_0 = (2, 2)$ and $\mathbf{b}_1 = (5, 2)$, contains for example the points $\mathbf{b}_1 - \mathbf{b}_0 = (3, 0)$ and $2\mathbf{b}_0 + 3\mathbf{b}_1 = (19, 10) \equiv (-5, -2) \bmod 12$.

The problem of finding the shortest non-zero vector in a lattice L given a basis for L, is called the Shortest Vector Problem (SVP). The smallest non-zero vectors in the lattice on the left in Figure 3.1 are $(1, -2)$ and $(-1, 2)$. For the lattice on the right, the smallest vectors are $(-1, 1)$ and $(1, -1)$. Let $\lambda_1(L)$ denote the length of the smallest non-zero vector in L. The approximate version of SVP consists of estimating $\lambda_1(L)$. The decisional approximate shortest vector problem ($\textsc{GapSVP}_\gamma$) is to decide whether $\lambda_1(L) \leq 1$ or $\lambda_1(L) > \gamma$, where $\gamma = \gamma(n)$ is some approximation factor. The Closest Vector Problem (CVP) is to find the lattice point that is closest to some given point that is not on the lattice. The computational hardness of these problems depends on the given basis vectors that define the lattice. Since the basis is not unique, any lattice can be defined by many different basis vectors. If the given lattice basis includes the shortest non-zero vector of this lattice, then SVP is trivial. If the given basis consists of large vectors (a *hard* basis) on the other hand, SVP becomes very hard to solve. It is conjectured that, given a hard basis, SVP and its variants cannot be solved for polynomial approximation factors in polynomial time. The LLL algorithm [76] can be used to find a relatively short basis for a lattice given by some hard basis. This algorithm runs in polynomial time but can only be used to solve the problem for subexponential approximation factors at best.

Let $\mathbf{s} \in \mathbb{Z}_q^n$ be some unknown vector and χ some distribution over $\mathbb{Z}_q$. Let $\mathbf{a}$ be some vector sampled from the uniform distribution over $\mathbb{Z}_q^n$, and $e \xleftarrow{\$} \chi(\mathbb{Z})$. An *LWE sample* for $\mathbf{s}$ is given by the pair $(\mathbf{a}, b)$, where $b = \mathbf{a}^\intercal \mathbf{s} + e$. The decisional LWE problem [100] is to distinguish between LWE samples and samples from the uniform random distribution over $\mathbb{Z}_q$. The search variant of the LWE problem consists of finding $\mathbf{s}$ given a number m of LWE samples for this vector. The problem can

Figure 3.2: The LWE problem in a lattice over $\mathbb{Z}_{12}$ of dimension 2: given some random basis $A = (\mathbf{a}_0, \mathbf{a}_1)$ and a point $\mathbf{b}$ close to the lattice, find $\mathbf{s}$ such that $A \cdot \mathbf{s}$ is close to $\mathbf{b}$.



be seen as solving a system of linear equations over $\mathbb{Z}_q$, in which only approximations of the true equations are given:

$$\mathbf{a}^{(1)} \cdot \mathbf{s}^{(1)} \approx \mathbf{b}^{(1)}$$
$$\vdots$$
$$\mathbf{a}^{(m)} \cdot \mathbf{s}^{(m)} \approx \mathbf{b}^{(m)}$$

Using matrix notation: given $\mathbf{A} \overset{\$}{\leftarrow} \mathbb{Z}_q^{n \times m}$ and an approximation $\mathbf{b}$ of the product $\mathbf{As}$, the LWE problem is to find $\mathbf{s}$. The approximation of $\mathbf{As}$ is determined by some *error vector* $\mathbf{e}$. This vector contains the χ-distributed values needed to obtain the exact equation $\mathbf{As} + \mathbf{e} = \mathbf{b}$. The LWE problem in dimension 2 is visualized in Figure 3.2.

The classical and quantum hardness of LWE and related problems have been studied extensively, after first results by [1] and [100] showed that these problems are suitable for use in post-quantum cryptography. Let parameters n, q and $\alpha \in (0, 1)$ such that $\alpha q > 2\sqrt{n}$, and χ a discrete Gaussian distribution with standard deviation αq. For these parameters, [100] showed that if there is an efficient quantum algorithm to solve this problem, then there is an efficient quantum algorithm to solve $\textsc{GapSVP}_\gamma$ on arbitrary lattices of dimension n with approximation factor $\tilde{O}(n/\alpha)$. This result implies that a random instance of LWE is as hard as a worst case instance of $\textsc{GapSVP}_\gamma$, i.e. given a hard basis. Since it is conjectured that worst case instances of $\textsc{GapSVP}_\gamma$ are hard to solve, there must be no efficient quantum algorithm to solve average case LWE instances. The fastest methods to solve LWE include the BKW algorithm [25], Babai's nearest plane algorithm [12], BKZ [107][37], sieving [2][19] and enumeration [55][57].

3.3.1 Cryptosystem

The first public-key encryption scheme based on the hardness of LWE was presented by Regev in [100]. The secret key is given by a uniform random vector $\mathbf{s} \in \mathbb{Z}_q^n$. The public key is a set of m LWE samples for this vector, given by $(\mathbf{A}, \mathbf{b} := \mathbf{A}\mathbf{s} + \mathbf{e})$, where $\mathbf{A}$ is sampled from the uniform distribution over $\mathbb{Z}_q^{n \times m}$ and the m coefficients of $\mathbf{e}$ are sampled from χ.

Algorithm 2 LWE-based Encryption [100]

```
    Input: Plaintext $\mu \in \{0,1\}$, public key $PK = (\mathbf{A}, \mathbf{b})$
    Output: Ciphertext $(\mathbf{c}_1, c_2)$
function $\textsc{Enc}(\mu, PK)$
    $\mathbf{v} \xleftarrow{\$} \{0,1\}^m$
    $\mathbf{c}_1 \leftarrow \mathbf{v}^\intercal \mathbf{A}$
    $c_2 \leftarrow \mathbf{v}^\intercal \mathbf{b} + \mu \left\lfloor \frac{q}{2} \right\rfloor$
```

Algorithm 3 LWE-based Decryption [100]

```
    Input: Secret key $SK = \mathbf{s}$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$
    Output: Plaintext $\mu$
function $\textsc{Dec}(C, SK)$
    $d \leftarrow c_2 - \mathbf{c}_1^\intercal \mathbf{s}$
    if d is closer to 0 than to $\left\lfloor \frac{q}{2} \right\rfloor$ then
        $\mu \leftarrow 0$
    else
        $\mu \leftarrow 1$
```

The value of $c_2 - \mathbf{c}_1^\intercal \mathbf{s}$ is close to $\mu \left\lfloor \frac{q}{2} \right\rfloor$, as can be seen by the following equations:

$$\begin{aligned} c_2 - \mathbf{c}_1^\intercal \mathbf{s} &= (\mathbf{A}\mathbf{s} + \mathbf{e})^\intercal \mathbf{v} + \mu \left\lfloor \frac{q}{2} \right\rfloor - \mathbf{v}^\intercal \mathbf{A}^\intercal \mathbf{s} \\ &= \mu \left\lfloor \frac{q}{2} \right\rfloor + \mathbf{e}^\intercal \mathbf{v} \\ &\approx \mu \left\lfloor \frac{q}{2} \right\rfloor, \end{aligned}$$

where the last approximation holds because the coefficients of $\mathbf{v}$ are binary and those of $\mathbf{e}$ are sufficiently small with high probability. This allows to decrypt the ciphertext using the secret key.

3.4 Ideal Lattices and RLWE

Regev's cryptosystem is not very efficient in terms of data transmission and computations per plaintext bit. To exchange one encrypted bit, $n+1$ coefficients in $\mathbb{Z}_q$ have to be sent. To increase the efficiency and reduce the size of the public key, one can add structure to the matrix $\mathbf{A}$. Let $\mathbf{a} = (a_0, \ldots, a_n)$ be a vector sampled from the uniform distribution over $\mathbb{Z}_q^n$. This vector will be the first row vector of $\mathbf{A}$. The $n-1$ remaining rows will be defined by applying an *anti-cyclic* shift to this row. This yields the following matrix:

$$\mathbf{A} := \begin{pmatrix} a_0 & a_1 & a_2 & \ldots & a_{n-2} & a_{n-1} \\ -a_{n-1} & a_0 & a_1 & \ldots & a_{n-3} & a_{n-2} \\ -a_{n-2} & -a_{n-1} & a_0 & \ldots & a_{n-4} & a_{n-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -a_1 & -a_2 & -a_3 & \ldots & -a_{n-1} & a_0 \end{pmatrix}$$

Let $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^n+1)$ and define the element $\mathbf{v} \in \mathcal{R}_q$ by $\mathbf{v} = \sum_{i=0}^{n-1} v_i x^i$ for each $\mathbf{v} \in \mathbb{Z}_q^n$. That is, each vector is considered to be a coefficient vector of an element in $\mathcal{R}_q$. Since $x^n \equiv -1 \bmod (x^n+1)$, applying an anti-cyclic shift to a coefficient vector $\mathbf{v}$ is equivalent to computing $x\mathbf{v} \bmod (x^n+1)$. The row vectors of the matrix $\mathbf{A}$ are obtained by computing $x^i\mathbf{a}$ in $\mathcal{R}_q$ for $0 \leq i < n$, so that $\mathbf{A} = (\mathbf{a}, x\mathbf{a}, x^2\mathbf{a}, \ldots, x^{n-1}\mathbf{a})^\intercal$. The multiplication $\mathbf{w}^\intercal\mathbf{A}$ can be written as

$$\begin{aligned} \mathbf{w}^\intercal\mathbf{A} &= \sum_{i=0}^{n-1} w_i \cdot x^i\mathbf{a} \bmod (x^n+1) \\ &= \mathbf{a}\sum_{i=0}^{n-1} w_i x^i \bmod (x^n+1) \\ &= \mathbf{aw} \bmod (x^n+1), \end{aligned}$$

that is, multiplication by an anti-cyclic matrix can be seen as multiplication in $\mathcal{R}_q$. The lattice generated by $\mathbf{A}$:

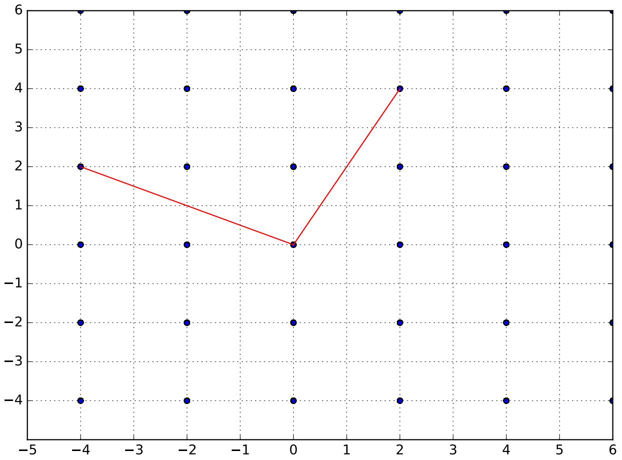
$$\mathcal{L} = \{\mathbf{w}^\intercal\mathbf{A} : \mathbf{w} \in \mathbb{Z}_q^n\}$$

is equivalent to the ideal in $\mathcal{R}_q$ generated by $\mathbf{a}$:

$$I = \{\mathbf{aw} \bmod (x^n+1) : \mathbf{w} \in \mathcal{R}_q\}.$$

These *ideal lattices* are central to the Ring-LWE (RLWE) problem and RLWE based cryptography. An example of an ideal lattice over $\mathbb{Z}_{12}[x]/(x^2+1)$ is given in Figure 3.3. It is generated by $\mathbf{a}(x) =$

Figure 3.3: Ideal lattice over $\mathbb{Z}_{12}[x]/(x^2+1)$ generated by $\mathbf{a}(x) := 2 + 4x$.



$2 + 4x$, corresponding to the basis vector $(2, 4)$. The second basis vector, $(-4, 2)$, is obtained by computing $x \cdot \mathbf{a}(x) \bmod (x^2 + 1) = 2x + 4x^2 \equiv -4 + 2x$. Every point on the lattice can be obtained by multiplying $\mathbf{a}$ by some polynomial in $\mathbb{Z}_{12}[x]/(x^2 + 1)$. Let $\mathbf{b} = 8 + 5x$ for instance. Then $\mathbf{ab} = 16 + 42x + 20x^2 \equiv -4 + 6x$. Therefore, $(-4, 6)$ is a lattice point.

Definition 1 (RLWE (informal version) [81])**.** *Let $\boldsymbol{s} \xleftarrow{\$} \mathcal{R}_q$ be uniformly random. RLWE samples for $\boldsymbol{s}$ are of the form $(\boldsymbol{a}, \boldsymbol{as} + \boldsymbol{e})$, where $\boldsymbol{a} \xleftarrow{\$} \mathcal{R}_q$ and the coefficients of $\mathbf{e}$ are drawn from some error distribution χ. The decisional version of RLWE is to distinguish between RLWE samples and uniformly random pairs $(\boldsymbol{v}, \boldsymbol{w})$ in $\mathcal{R}_q$.*

It is shown by [81] that a generalized version of this problem is computationally hard, on the condition that it is hard for any polynomial-time quantum algorithm to approximate the search version of SVP in the worst case on ideal lattices. While there exist quantum algorithms solving SVP over ideal lattices by exploiting the algebraic structure of ideal lattices [43], their approximation factor is too large to affect the security of RLWE-based cryptography. The RLWE problem remains hard when the secret polynomial $\mathbf{s}$ is sampled from the error distribution. The RLWE based cryptosystem defines the secret key $\mathbf{s} \xleftarrow{\$} \chi(\mathcal{R}_q)$, and public keys $(\mathbf{a}, \mathbf{b} := \mathbf{as} + \mathbf{e})$, where $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ and $\mathbf{e} \xleftarrow{\$} \chi(\mathcal{R}_q)$. Encryption and decryption are defined by Algorithms 4 and 5.

3.4.1 CPA and CCA Security

The cryptosystem described by algorithms 4 and 5 is secure in the Chosen Plaintext Attack (CPA) model. In this attack model, the adversary is given two plaintexts and a ciphertext that encrypts one of the two, and must determine which plaintext is encrypted by the ciphertext. To do this, the adversary may query an encryption oracle. If such an adversary cannot succeed, then the cryptosystem is called CPA-secure (or IND-CPA). The RLWE encryption scheme essentially *masks* the plaintext using an

Algorithm 4 RLWE Encryption [81]

```
Input: Plaintext $\mu \in \{0,1\}^n$, Public key $(\mathbf{a},\mathbf{b}) \in \mathcal{R}_q^2$
Output: Ciphertext $(\mathbf{c}_1,\mathbf{c}_2) \in \mathcal{R}_q^2$
function Enc($\mu, \mathbf{a}, \mathbf{b}$)
    $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \xleftarrow{\$} \chi(\mathcal{R}_q)$
    View $\mu$ as a polynomial in $\mathcal{R}_q$
    $\mathbf{c}_1 \leftarrow \mathbf{a}\mathbf{e}_1 + \mathbf{e}_2$
    $\mathbf{c}_2 \leftarrow \mathbf{b}\mathbf{e}_1 + \mathbf{e}_3 + \left\lfloor \frac{q}{2} \right\rfloor \mu$
```

Algorithm 5 RLWE Decryption [81]

```
Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1,\mathbf{c}_2) \in \mathcal{R}_q^2$
Output: Plaintext $\mathbf{d} \in \{0,1\}^n$
function Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$
    for all coefficients $d$ of $\mathbf{d}$ do
        if d is closer to 0 than to $\left\lfloor \frac{q}{2} \right\rfloor$ then
            $d \leftarrow 0$
        else
            $d \leftarrow 1$
```

RLWE sample. Since samples from the RLWE distribution are indistiguishable from pseudorandom samples, the ciphertext is pseudorandom. It is therefore not possible for the adversary, given two plaintext and a ciphertext, to determine which plaintext is encrypted by the given ciphertext. The RLWE-scheme is CPA-secure.

In practice, security in the CPA model may not be sufficient, as it assumes a fairly weak attacker. The Chosen Ciphertext Attack (CCA) model assumes a more powerful attacker, who has access to a decryption oracle. The adversary may send any query to the decryption oracle, except for the given ciphertext. The goal, again, is to choose the correct plaintext given two plaintexts and an encryption of one the two. To succeed in this goal, the adversary could try to find the secret key using the decryption oracle. Let $(\mathbf{c}_1, \mathbf{c}_2)$ be a ciphertext such that all the coefficients are equal to zero, except for the first coefficient of $\mathbf{c}_1$ which is equal to -1. Then algorithm 5 computes $\mathbf{d} \leftarrow 0 - (-1) \cdot \mathbf{s}$, which is equal to the secret key. For each coefficient s_i of $\mathbf{s}$, the output of the decryption oracle reveals whether s_i is closer to 0 than to $\left\lfloor \frac{q}{2} \right\rfloor$. By making multiple queries and varying the coefficients of $\mathbf{c}_2$, the adversary can thus recover the complete secret key. Therefore, algorithm 5 is not secure in the CCA model.

The Fujisaki-Okamoto transform [56] is a generic method to convert CPA-secure encryption and decryption algorithms to a CCA-secure Key-Exchange Mechanism (KEM). The transform uses some cryptographic hash functions G and H, and is described by algorithms 6 and 7. In order to prevent CCA on the decryption function, the decrypted ciphertext is re-encrypted and compared to the received ciphertext. This allows to verify that the ciphertext is valid, that is, it was generated using the encryption function. Chosen ciphertext attacks like the one described above are prevented by returning a string of random bits if the re-encryption is not equal to the received ciphertext. One technicality remains however, since the encryption function in non-deterministic. In order to make the CCA-secure encapsulation deterministic, the source of randomness used during the encryption is uniquely determined by evaluating a hash function on the public key and the plaintext. This ensures that a re-encryption of the plaintext can be correctly reconstructed using the decrypted ciphertext. To the contrary of the CPA-secure scheme, the plaintext in the CCA-secure scheme is not the same as the session key. The session key is created by hashing public key and plaintext dependent data together with the ciphertext.

Algorithm 6 CCA-secure Encapsulation [56]

```
Input: Public key $PK$, random $\mu \in \{0,1\}^n$
Output: Ciphertext $C$ and session key $K$
function ENCAPS( )
    $(r, d) \leftarrow H(PK||\mu)$
    $C \leftarrow \textsc{Enc}(\mu, PK, r)$
    $K \leftarrow G(C||d)$
```

Algorithm 7 CCA-secure Decapsulation [56]

```
Input: Ciphertext $C$, Public key $PK$ and secret key $SK$
Output: Session key $K'$
function DECAPS( )
    $\mu' \leftarrow \textsc{Dec}(C, SK)$
    $(r', d') \leftarrow H(PK||\mu')$
    $C' \leftarrow \textsc{Enc}(\mu', PK, r')$
    if $C' == C$ then
        $K' \leftarrow G(C'||d')$
    else
        $K' \xleftarrow{\$} \{0,1\}^{256}$
```

3.4.2 Generalization and Module LWE

Module LWE (MLWE) was introduced by [75]. It is a generalization of RLWE and uses small matrices and vectors over the polynomial ring $\mathcal{R}_q$.

Definition 2 (MLWE [75])**.** *For some integer parameter $k > 0$, an MLWE sample for some secret vector of polynomials $\mathbf{s} \in \mathcal{R}_q^k$ is given by some uniformly random vector $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q^k$, together with the polynomial $\mathbf{b} = \mathbf{a}^\intercal \mathbf{s} + \mathbf{e}$, where $\mathbf{e} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q)$. The search MLWE problem is to find $\mathbf{s}$ given a number of samples.*

Table 3.1: Parameter sets to distinguish between LWE, RLWE, and MLWE based cryptosystems. The number of rows and columns indicated are for the secret key matrix/vector/polynomial.

algorithm	polynomial degree n	number of columns m	number of rows k	modulus q
LWE	1	small, > 1	large	large[1]
RLWE	large[1]	1	1	large [2]
MLWE	large[1]	1	small, > 1	large [2]

Note that for the polynomial degree $n = 1$ the MLWE problem is similar to the LWE problem with vectors of length k. For $k = 1$, MLWE is equivalent to RLWE over the ring $\mathcal{R}_q$. In the MLWE variant, small matrices and vectors with polynomial coefficients are used. Table 3.1 shows how parameters n, k and m allow to distinguish between LWE, RLWE and MLWE. It has been shown by [75] that MLWE is at least as hard as solving some hard lattice problems using quantum algorithms.

Algorithms 8 and 9 describe the framework used for instance by NewHope, Kyber and FrodoKEM. Variations of this framework include the use of deterministic errors ("Learning with Rounding", LWR) [45] or using Gaussian noise (used in FrodoKEM) instead of sampling the binomial distribution. The private key is defined by sampling some $\mathbf{s} \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{k\times m})$. Then the public key is determined by computing a number of LWE/RLWE/MLWE samples for this secret. That is, sample a uniform random $\mathbf{A} \xleftarrow{\$} \mathcal{R}_q^{k\times k}$ and $\mathbf{e}_0 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{m\times k})$. The public key is given by $(\mathbf{A}, \mathbf{b})$ where $\mathbf{b} := \mathbf{s}^\intercal\mathbf{A} + \mathbf{e}_0$.

Algorithm 8 Encryption [27]

```
    Input: Plaintext $\mu \in \{0, \dots, 2^B\}^{m^2 n}$, $PK = (\mathbf{A}, \mathbf{b})$
    Output: Ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$
function Enc($\mu, PK$)
    $\mathbf{e}_1, \mathbf{e}_2 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{k\times m})$
    $\mathbf{e}_3 \xleftarrow{\$} \mathcal{B}_\lambda(\mathcal{R}_q^{m\times m})$
    $\mathbf{c}_1 \leftarrow \mathbf{e}_1^\intercal\mathbf{A} + \mathbf{e}_2^\intercal$
    $\mathbf{c}_2 \leftarrow \mathbf{b}\mathbf{e}_1 + \mathbf{e}_3 + \text{ENCODE}_B(\mu)$
```

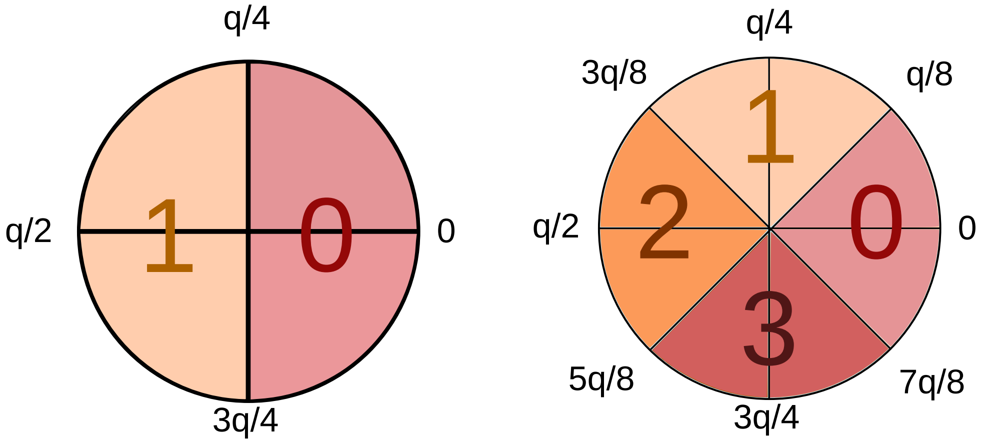
Algorithm 9 Decryption [27]

```
    Input: Secret key $SK = \mathbf{s}$, ciphertext $C = (\mathbf{c}_1, \mathbf{c}_2)$
    Output: Plaintext $\mu$
function Dec($C, SK$)
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$
    $\mu \leftarrow \text{DECODE}_B(\mathbf{d})$
```

[1] For computational reasons such as simplifying modular reduction or using the NTT, these parameters are often chosen to be powers of 2.

[2] These parameters need to be prime numbers in order to use the NTT for polynomial multiplication.

Figure 3.4: Left: Decoding the coefficients when $B = 0$. Coefficients closer to 0 than to $\lfloor \frac{q}{2} \rceil$ are mapped to 0, while all the other coefficients are mapped to 1. Right: decoding when $B = 1$.



The number of bits encoded in each plaintext coefficient is equal to $B+1$. For RLWE and MLWE, the parameter B is set to zero and $\textsc{encode}_B(\mu)$ lifts μ to the ring $\mathcal{R}_q$ in a straightforward coefficient-wise manner and returns $\mu \lfloor \frac{q}{2} \rceil$. The $\textsc{decode}_B(\mathbf{d})$ function maps coefficients of $\mathbf{d}$ to 0 if they are in the interval $\{\lfloor \frac{-q}{4} \rceil, \ldots, \lfloor \frac{q}{4} \rceil\}$, else they are mapped to 1 (as depicted on the left in Figure 3.4). In LWE each coefficient encodes a number of bits $B \geq 1$. Encoding then lifts μ to the module $\mathbb{Z}_q^{m \times m}$ and involves a scalar multiplication by $\lfloor \frac{q}{2^{B+1}} \rceil$. Decoding is generalized by dividing $\mathbb{Z}_q$ up into 2^{B+1} intervals as shown on the right in Figure 3.4.

For $n > 1$ and $k = m = 1$, Algorithms 2 and 3 describe the RLWE-based encryption scheme. Ciphertexts, plaintexts and keys are then polynomials in $\mathcal{R}_q$. For $n = 1$ and $k, m > 1$ the ring $\mathcal{R}_q$ is equal to $\mathbb{Z}_q$ and the plain LWE scheme is obtained, with ciphertexts, plaintexts and keys in the form of matrices over $\mathbb{Z}_q$. The intermediate parameter sets for which $n, k > 1$ define the MLWE variant of the scheme.

If $k = 2$, for example, an MLWE secret key is a vector $\mathbf{s} = (\mathbf{s}_1(x), \mathbf{s}_2(x))^\intercal$, where the coefficients of both polynomial are sampled from the uniform distribution over $\mathbb{Z}_q$. To create a public key, four uniformly random (in $\mathcal{R}_q$) polynomials $\mathbf{a}_{00}(x)$, $\mathbf{a}_{01}(x)$, $\mathbf{a}_{10}(x)$ and $\mathbf{a}_{11}(x)$ must be sampled, and two error polynomials $\mathbf{e}_0(x), \mathbf{e}_1(x)$ are sampled from $\mathcal{B}_\lambda(\mathcal{R}_q)$. Then the public vector $\mathbf{b} = (\mathbf{b}_0(x), \mathbf{b}_1(x))$ is computed as follows:

$$\begin{pmatrix} \mathbf{b}_0(x) & \mathbf{b}_1(x) \end{pmatrix} = \begin{pmatrix} \mathbf{s}_1(x) & \mathbf{s}_1(x) \end{pmatrix} \begin{pmatrix} \mathbf{a}_{00}(x) & \mathbf{a}_{01}(x) \\ \mathbf{a}_{10}(x) & \mathbf{a}_{11}(x) \end{pmatrix} + \begin{pmatrix} \mathbf{e}_1(x) & \mathbf{e}_1(x) \end{pmatrix} \tag{3.1}$$

An important advantage of MLWE-based cryptosystems is the flexibility in choosing parameters, while still benefiting from fast arithmetic in $\mathcal{R}_q$ using the NTT. In RLWE, efficient use of the NTT means that the degree n has to be a power of 2, restricting the lattice dimension to a very limited subset. In MLWE one would also fix n to be a power of 2, but the total dimension of the lattice is determined by $n \times k$, allowing more choices. To instantiate the MLWE scheme for a higher security level, it suffices to increase the vector size k. In [27] the MLWE-based Kyber key-exchange mechanism is defined for $k = 2, 3$ and 4, where $k = 4$ aims for the highest level of security, equivalent to AES-256.

3.4.3 NTRU

Similar to RLWE-based PKE, the NTRU cryptosystem [59] is defined over a polynomial ring $\mathcal{R} = \mathbb{Z}[x]/(x^n - 1)$. Let q be some integer coprime with 3 and such that $q > 3$. The secret key is some $\mathbf{f} \in \mathcal{R}$ with small coefficients that is inversible modulo 3 and modulo q. A random polynomial $\mathbf{g} \in \mathcal{R}$ with small coefficients is used to compute $\mathbf{h} = (\mathbf{f} \bmod q)^{-1}\mathbf{g}$. This $\mathbf{h}$ will be the public key. Encryption and decryption are defined in Algorithm 10.

Algorithm 10 NTRU encryption and decryption [59]

```
Encryption of $\mu \in \mathcal{R}$ with coefficients in $\{-1, 0, 1\}^n$ using public key $PK = \mathbf{h}$.
function Enc($\mu, PK$)
    $\phi \xleftarrow{\$} \mathcal{R}$ with small coefficients
    $\mathbf{c} \leftarrow 3\phi \cdot \mathbf{h} + \mu \bmod q$
    return $\mathbf{c}$
Decryption of ciphertext $\mathbf{c} \in \mathcal{R}$ using secret key $SK = \mathbf{f}$.
function Dec($\mathbf{c}, SK$)
    $\mathbf{a} \leftarrow \mathbf{f} \cdot \mathbf{c} \bmod q$
    return $(\mathbf{f} \bmod 3)^{-1} \cdot \mathbf{a} \bmod 3$
```

3.4.4 LWR

The use of the NTT restricts the choice of the parameters n and q for the ring $\mathbb{Z}_q[x]/(x^n + 1)$. The degree n must be a power of 2. In practice, one would want to instantiate an RLWE cryptosystem for some $2^9 < n < 2^{10}$. An encryption scheme for $2^9 = 512$ would be considered not secure enough, while $n = 2^{10}$ would be overkill. Parameters used for NTRU schemes include for example $n = 653, 761$ and 857 [21]. Flexibility in parameters is one of the reasons why some RLWR/MLWR schemes such as Round5 and Saber choose alternative multiplication methods over the NTT. The transmission of keys and ciphertexts consisting of polynomials of degree 700 is cheaper than the transmission of polynomials of degree 1024. This choice also allows them to use a power of 2 modulus q, so that modular reduction needed for arithmetic in $\mathbb{Z}_q$ is trivial to compute. When using a power of 2 modulus, it is possible to simplify the error sampling process. Instead of sampling from an error distribution and adding the error to the ciphertext during encryption, a number of least significant bits of each coefficient of the ciphertext is removed, thereby "rounding" each coefficient. The security of cryptosystems using this technique is based on the hardness of the Learing With Rounding (LWR) problem, and its ring (RLWR) and module (MLWR) variants. The rounding function denoted $\lfloor \cdot \rceil_p : \mathbb{Z}_q \to \mathbb{Z}_p$ is defined for

$p < q$ by:

$$\lfloor a \rceil_p = \lfloor (p/q) \rceil \, a \bmod p \tag{3.2}$$

If p and q are both powers of 2, then this rounding function corresponds to removing the $\log_2(q) - \log_2(p)$ least significant bits. For some secret polynomial $\mathbf{s} \in \mathcal{R}_q$, an RLWR sample is generated by sampling an uniform random $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ and returning $\mathbf{a}$ and $\lfloor \mathbf{ab} \rceil_p$. This problem was first studied in [14]. MLWR/RLWR based cryptosystems include Round5 [11] and Saber [45].

3.5 Implementation of LWE-based Cryptography

3.5.1 Modular arithmetic

In lattice cryptography computations are often performed in a fixed finite ring. There is no need for a general modular reduction algorithm that works for any modulus. Specialized reduction algorithms for some fixed modulus can be used to speed up the modular arithmetic.

Barrett reduction

A modular reduction $x \bmod q$ can be seen as the computation of $x - \left\lfloor \frac{x}{q} \right\rfloor q$. This computation involves a floating-point division. To avoid this costly operation, the Barrett reduction algorithm [17] uses a precomputed constant that is determined by a given modulus. Let q be a modulus that is not a power of 2 and $w = \lceil \log_2 q \rceil$ the length of its binary representation. The Barrett algorithm precomputes $r = \left\lfloor \frac{2^{2w}}{q} \right\rfloor$ and reduces any integer $x < q^2$ with the following procedure:

1. Let $p \leftarrow r \cdot x$.
2. Shift p to the right by $2w$ positions, keeping only the integer part.
3. $\bar{x} \leftarrow x - p \cdot q$.
4. If $\bar{x} < q$ return $\bar{x}$, else return $\bar{x} - q$.

Two multiplications and a subtraction in $\mathbb{Z}$ suffice to obtain the modular reduction. A different method to compute modular reduction using two multiplications is given by the Montgomery reduction algorithm [88].

For some specific moduli, even these multiplications can be simplified. A multiplication of some x by $y = 2^{l_1} + 2^{l_2} + 1$ for some integers $l_1, l_2 > 0$ can be written as:

$$\begin{aligned} yx &= 2^{l_1}x + 2^{l_2}x + x \\ &= (x << l_1) + (x << l_2) + x, \end{aligned}$$

where the << operater denotes a leftshift of the bits. This method of multiplication is particularly interesting when multiplying by a constant of the form

$$2^{l_M} \pm 2^{l_{M-1}} \pm \cdots \pm 2^{l_1}, \tag{3.3}$$

where M is a small integer. The Barrett reduction algorithm consists of two multiplications by constants: the modulus n and the precomputed factor r. If both constants are of the form (3.3) it might be preferable to replace the multiplications by bit-wise shifts and additions. An example is given by the modulus $8380417 = 2^{23} - 2^{13} + 1$ and its corresponding precomputation constant

$$r = \left\lfloor \frac{2^{46}}{8380417} \right\rfloor = 8396807 = 2^{23} + 2^{13} + 2^3 - 1.$$

This modulus is used in the Dilithium signature scheme [50]. In [78] an adaptation of the Barrett algorithm is described. The efficiency of their algorithm also depends on the fact that for $q = 7681$ the Barrett constant $\left\lfloor \frac{2^{2l_1}}{q} \right\rfloor$ has a short binary representation. Modular reduction can then be computed in just a few bit-wise shifts and additions.

Modular reduction for $q = 2^{l_1} - 2^{l_2} + 1$ In the LWE scheme the modulus is a power of 2, so that no computation is required to compute modular reduction. In order to compute $a \bmod 2^R$ for some $2^R < a < 2^{2R}$ and some exponent R, note that a can be written as $a = a_0 + 2^R a_1$ for some $a_0, a_1 < 2^R$. Then $a \bmod 2^R \equiv a_0$. In other words, modular reduction for moduli of the form 2^R, is computed by simply taking the R least significant bits of a.

In RLWE/MLWE however, in order to use the NTT, the existence of a $2n$-th root of unity in $\mathbb{Z}_q$ is required. This is the case if q is a prime for which $q \equiv 1 \bmod 2n$. The choices for q are therefore limited. For prime moduli of the form $q = 2^{l_1} - 2^{l_2} + 1$ for some integers l_1 and l_2, one has $2^{l_1} - 2^{l_2} + 1 \equiv 1 \bmod 2n$ if $l_2 \geq \log_2(2n)$. For $n = 256$ suitable primes include $7681 = 2^{13} - 2^9 + 1$ which is used in the original version of Kyber and for $n = 1024$ the prime $q = 2^{14} - 2^{12} + 1 = 12289$ is used in NewHope. A modular reduction method in the style of [109] can be used for moduli of the form $2^{l_1} - 2^{l_2} + 1$. Using the fact that $2^{l_1} \equiv 2^{l_2} - 1 \bmod q$, a modular reduction can be computed using only bitwise shifts, additions and subtractions.

3.5.2 Polynomial arithmetic

NTT

Let q be a prime number and n power of 2 such that $q \equiv 1 \bmod 2n$. A multiplicative generator for $\mathbb{Z}_q$ is an element $g \in \mathbb{Z}_q$ such that $g^i \bmod q \neq 1$ for all $i < q - 1$. Then the element $\phi = g^{(q-1)/2n} \bmod q$ is a primitive $2n$-th root of unity in $\mathbb{Z}_q$. Using the n-th root of unity $\omega = \phi^2$, the Number Theoretic Transform (NTT) can be computed.

Definition 3 (NTT [95])**.** *Let $\omega \in \mathbb{Z}_q$ be a primitive n-th root of unity and $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i x^i$ an element in $\mathcal{R}_q$. Then the NTT is defined by the map from $\mathcal{R}_q$ to $\mathcal{R}_q$:*

$$\boldsymbol{a}(x) \mapsto \sum_{j=0}^{n-1} \boldsymbol{a}(\omega^j) x^j \tag{3.4}$$

The inverse of the NTT is denoted NTT^{-1}. It is defined using the inverse $\omega^{-1} \bmod q$ and $n^{-1} \bmod q$, by the map:

$$\mathbf{a}(x) \mapsto n^{-1} \sum_{j=0}^{n-1} \mathbf{a}(\omega^{-j}) x^j \tag{3.5}$$

To verify that this is indeed the inverse of the NTT, consider for some polynomial $\mathbf{a} = \sum_{k=0}^{n-1} a_k x^k$:

$$\begin{aligned} \text{NTT}^{-1}(\text{NTT}(\mathbf{a})) &= n^{-1} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k \omega^{kj} \right) \omega^{-ij} x^i \\ &= n^{-1} \sum_{i=0}^{n-1} x^i \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} \omega^{j(k-i)} \end{aligned} \tag{3.6}$$

The expression $\sum_{j=0}^{n-1} \omega^{j(k-i)}$ is equal to 0 for all $k \neq i$, while equal to $\sum_{j=0}^{n-1} 1$ for $k = i$. Then from equation (3.6) follows:

$$\begin{aligned} \text{NTT}^{-1}(\text{NTT}(\mathbf{a})) &= n^{-1} \sum_{i=0}^{n-1} x^i a_i \sum_{j=0}^{n-1} 1 \\ &= n^{-1} \sum_{i=0}^{n-1} x^i a_i n = \sum_{i=0}^{n-1} a_i x^i = \mathbf{a} \end{aligned}$$

To use the NTT for multiplication in $\mathcal{R}_q$, the polynomials have to be pre-processed using the *negative wrapped convolution* (NWC) [80]. Let $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathcal{R}_q$ such that

$$\mathbf{a}(x)\mathbf{b}(x) = \mathbf{c}(x) + \mathbf{d}(x)(x^n + 1) \tag{3.7}$$

in $\mathbb{Z}_q[x]$, where $\mathbf{c}(x)$ is a polynomial of degree smaller than n. Then $\mathbf{c} = \mathbf{ab} \bmod (x^n + 1)$. Let $\phi \in \mathbb{Z}_q$ be a primitive $2n$-th root of unity such that $\phi^n = -1 \bmod q$. Then, in $\mathbb{Z}_q[x]$ and for $i \geq 0$, one has:

$$\begin{aligned}
\mathbf{a}(\phi\omega^i)\mathbf{b}(\phi\omega^i) &\equiv \mathbf{c}(\phi\omega^i) + \mathbf{d}(\phi\omega^i)((\phi\omega^i)^n + 1) \bmod q \\
&\equiv \mathbf{c}(\phi\omega^i) + \mathbf{d}(\phi\omega^i)(-1+1) \bmod q \\
&= \mathbf{c}(\phi\omega^i)
\end{aligned}$$

This means that $\mathrm{NTT}(\mathbf{a}(\phi x)) \odot \mathrm{NTT}(\mathbf{b}(\phi x)) = \mathrm{NTT}(\mathbf{c}(\phi x))$. Using the pointwise multiplication "$\odot$", polynomial multiplications may be computed in the NTT domain instead of computing them in the time domain. First the NTTs of both input polynomials have to be computed, then the pointwise multiplication, and finally the result has to be mapped back to the time domain. The pointwise multiplication "$\odot$" consists of only n independent multiplications in $\mathbb{Z}_q$. Computing polynomial products in the NTT domain is therefore much less costly than multiplication in the time domain, on the condition that the NTT itself can be computed efficiently.

Also note that the reduction $\bmod(x^n + 1)$ is obtained for free by using the NTT and the NWC. To obtain the correct result from the polynomial multiplication, the inverse of the NWC should be applied to $\mathrm{NTT}^{-1}(\mathrm{NTT}(\mathbf{c}))$. That is, each coefficient has to be multiplied by a power of ϕ^{-1}.

Computation of the NTT

The NTT can be efficiently computed using the Cooley-Tukey [40] algorithm. This algorithm recursively expresses an n-point NTT into 2 $n/2$-point NTTs. Using definition 3, the k-th coefficient of the image $\hat{\mathbf{a}} = \mathrm{NTT}(\mathbf{a})$ can be written as:

$$\hat{a}_k = \mathbf{a}(\omega^k) = \sum_{i=0}^{n-1} a_i \omega^{ik} \tag{3.8}$$

Separating the even powers from the odd powers, we get:

$$\begin{aligned}
\hat{a}_k &= \sum_{i=0}^{n/2-1} a_{2i}\omega^{2ik} + \sum_{i=0}^{n/2-1} a_{2i+1}\omega^{(2i+1)k} && (3.9) \\
&= \sum_{i=0}^{n/2-1} a_{2i}\omega^{2ik} + \omega^k \sum_{i=0}^{n/2-1} a_{2i+1}\omega^{2ik} && (3.10) \\
&= \mathcal{X}_k + \omega^k \mathcal{Y}_k, && (3.11)
\end{aligned}$$

where $\mathcal{X}_k := \sum_{i=0}^{n/2-1} a_{2i}\omega^{2ik}$ and $\mathcal{Y}_k := \sum_{i=0}^{n/2-1} a_{2i+1}\omega^{2ik}$. Then $\hat{a}_{k+n/2}$ can be expressed in $\mathcal{X}_k$ and

$\mathcal{T}_k$, using the fact that $\omega^{n/2} = -1 \bmod q$:

$$\begin{aligned}\hat{a}_{k+n/2} &= \sum_{i=0}^{n/2-1} a_{2i}\omega^{2i(k+n/2)} + \omega^{k+n/2} \sum_{i=0}^{n/2-1} a_{2i+1}\omega^{2i(k+n/2)} \\ &= \sum_{i=0}^{n/2-1} a_{2i}\omega^{2ik} + \omega^k\omega^{n/2} \sum_{i=0}^{n/2-1} a_{2i+1}\omega^{2ik} \\ &= \mathcal{X}_k - \omega^k\mathcal{Y}_k.\end{aligned}$$

This means that 2 coefficients of NTT($\mathbf{a}$) can be computed using $\mathcal{X}_k$ and $\mathcal{Y}_k$. Assuming that n is a power of 2, then the same method can be applied to summations $\mathcal{X}_k$ and $\mathcal{Y}_k$. They can be split into subsequences of even and odd powers, of length $n/4$ respectively. The Cooley-Tukey algorithm consists of the divide-and-conquer strategy that continues this process upon obtaining subsequences of length 1.

Other multiplication methods

Other polynomial multiplication methods include schoolbook, Karatsuba [68] or specialized algorithms for sparse polynomial multiplication. The schoolbook method takes two polynomials $\mathbf{a} = \sum_{i=0}^{n-1} a_i x^i$ and $\mathbf{b} = \sum_{i=0}^{n-1} b_i x^i$ and computes the product $\mathbf{ab} = \mathbf{c} = \sum_{i=0}^{2n-2} c_i x^i$, where

$$c_i = \sum_{k \leq i} a_k b_{i-k} \tag{3.12}$$

Each coefficient of $\mathbf{a}$ is multiplied by each coefficient of $\mathbf{b}$. In order to obtain the full product $\mathbf{ab}$, exactly n^2 multiplications in $\mathbb{Z}_q$ have to be computed.

This of operations number can be reduced using the Karatsuba algorithm. A polynomial of degree n can be expressed as a function of two degree $\frac{n}{2}$ polynomials, assuming that n is even:

$$\begin{aligned}\mathbf{a} &= \sum_{i=0}^{n-1} a_i x^i \\ &= \sum_{i=0}^{n/2-1} a_i x^i + x^{n/2} \sum_{j=n/2}^{n-1} a_j x^j \\ &=: \mathbf{a}^{(0)} + x^{n/2}\mathbf{a}^{(1)}\end{aligned}$$

for some polynomials $\mathbf{a}^{(0)}$ and $\mathbf{a}^{(1)}$ of degree at most $n/2$. Similarly, let $\mathbf{b} = \mathbf{b}^{(0)} + x^{n/2}\mathbf{b}^{(1)}$. In order to compute the product

$$
\begin{aligned}
\mathbf{ab} &= (\mathbf{a}^{(0)} + x^{n/2}\mathbf{a}^{(1)})(\mathbf{b}^{(0)} + x^{n/2}\mathbf{b}^{(1)}) \\
&= \mathbf{a}^{(0)}\mathbf{b}^{(0)} + x^{n/2}(\mathbf{a}^{(0)}\mathbf{b}^{(1)} + \mathbf{a}^{(1)}\mathbf{b}^{(0)}) + \mathbf{a}^{(1)}\mathbf{b}^{(1)},
\end{aligned}
$$

4 partial products have to be computed: $\mathbf{a}^{(0)}\mathbf{b}^{(0)}, \mathbf{a}^{(0)}\mathbf{b}^{(1)}, \mathbf{a}^{(1)}\mathbf{b}^{(0)}$ and $\mathbf{a}^{(1)}\mathbf{b}^{(1)}$. The Karatsuba method only computes 3 partial products. First $\mathbf{a}^{(0)}\mathbf{b}^{(0)}$ and $\mathbf{a}^{(1)}\mathbf{b}^{(1)}$ are computed, and then the $x^{n/2}$ term $(\mathbf{a}^{(0)}\mathbf{b}^{(1)} + \mathbf{a}^{(1)}\mathbf{b}^{(0)})$ is computed directly using the identity:

$$
(\mathbf{a}^{(0)} - \mathbf{a}^{(1)})(\mathbf{b}^{(0)} - \mathbf{b}^{(1)}) = \mathbf{a}^{(0)}\mathbf{b}^{(0)} - (\mathbf{a}^{(0)}\mathbf{b}^{(1)} + \mathbf{a}^{(1)}\mathbf{b}^{(0)}) + \mathbf{a}^{(1)}\mathbf{b}^{(1)}.
$$

The third partial product to be computed is therefore $(\mathbf{a}^{(0)} - \mathbf{a}^{(1)})(\mathbf{b}^{(0)} - \mathbf{b}^{(1)})$. The Karatsuba algorithm thus computes a product of two degree n polynomials by computing 3 partial products of degree $n/2$ polynomials. The fourth partial product can be omitted at the cost of some additional subtractions. If n is a power of 2, then the same process can be repeatedly applied to the partial products. The complexity, expressed in integer multiplications, decreases asymptotically from $\mathcal{O}(n^2)$ (for schoolbook multiplication) to $\mathcal{O}(n^{\log_2(3)})$.

Sparse multiplication methods can be used in the case that the number of non-zero coefficients of one of the operands is small. This means that a straightforward computation using the schoolbook algorithm would spend most of its time multiplying zeroes. To avoid this, it makes sense to store only the indices of non-zero coefficients and their values. This technique is typically used in NTRU implementations (see for instance [13]).

3.5.3 Lattice Cryptography on FPGA

Table 3.2 shows implementation results of state of the art FPGA implementations of lattice based public key encryption. Polynomial multiplication is the most time consuming arithmetic operation in RLWE and MLWE based cryptosystems. Different solutions have been proposed in the state of the art FPGA implementations. The schoolbook algorithm is implemented by [97] and [77], in order to minimize the resource utilization. These area-optimized implementations are much slower than NTT-based solutions such as the one proposed by [105]. Comparing the computation time of the encryption algorithm of [97] and [105], it can be seen that the NTT-based implementation is around 50 times faster. The area results on the other hand, are less clear. While the schoolbook based solution [97] uses 4 times less LUTs, the DSP and BRAM utilization is the same. It seems that the trade-off between area and computation time is in favor of the NTT. It should be mentioned though, that [105] uses a Virtex-6 FPGA, which is better and more expensive than the Spartan-6 used by [97]. The very recent RLWE-256 implementation by [116] uses schoolbook polynomial multiplication. Their computation time is more than six times as high as the one reported in 2014 by [105], using the NTT. Moreover, the schoolbook implementation uses one more DSP block, and has a very similar LUT utilization

Table 3.2: CPA and CCA-secure Encryption or Encapsulation (CPA, CCA) or 'Server' part in Client-Server-Client key exchange (K-E), from the state of the art. If marked with (*), resource results are for both encryption and decryption.

				Freq.	Time	Area
Src.	Algorithm	Type	FPGA	MHz	μ s	DSP, BRAM, Slice, LUT
[97]	RLWE-256	CPA	xc6slx9	128	1070	1, 2, 114, 360
[97]	RLWE-256	CPA	xc6slx9	144	946	1, 2, 95, 282
[96]	RLWE-256	CPA	xc6slx16	160	43	1, 14, n.a., 4121 (*)
[96]	RLWE-256	CPA	xc6vlx75t	262	26	1, 12, n.a., 4549 (*)
[105]	RLWE-256	CPA	xc6vlx76	313	20	1, 2, n.a., 1349 (*)
[77]	RLWE-256	CPA	Kintex-7	305	229	1, 3, 303, 898
[116]	RLWE-256	CPA	Kintex-7	280	128	2, 2, 402, 1254
[116]	RLWE-256	CPA	Kintex-7	275	129	2, 2, 479, 1381 (*)
[74]	RLWE-1024	K-E	xc7z020	131	79	8, 14, n.a. 20826
[90]	RLWE-1024	K-E	xc7a35t	117	1532	2, 4, n.a., 4498
[5]	RLWR-1170	CCA	5csema	130	1350	6337 ALM, 11765 bytes (*)
[5]	RLWR-1018	CPA	5csema	133	1000	4116 ALM, 10753 bytes (*)
[44]	RLWR-1170	CCA	xczu9eg	212	30	0, 4, 18733, 91166 (*)
[115]	RLWE-1024	CCA	xc7z020	200	62	2, 8, n.a, 6781 (*)
[63]	LWE-256	CPA	xc6slx45	125	786	1, 73, 1866, 6152
[64]	LWE-640	CCA	Artix-7	183	4624	4, 0, 1338, 4620
[64]	LWE-640	CCA	Artix-7	177	2342	8, 0, 1485, 5155
[64]	LWE-640	CCA	Artix-7	171	1212	16, 0, 1692, 5796
[65]	LWE-640	CCA	xc7a35t	167	19608	1, 11, 1855, 6745
[44]	LWE-640	CCA	xczu9eg	402	352	32, 27, 1186, 7213 (*)
[64]	LWE-976	CCA	Artix-7	180	10638	4, 0, 1455, 4996
[64]	LWE-976	CCA	Artix-7	175	5464	8, 0, 1608, 5562
[64]	LWE-976	CCA	Artix-7	168	2857	16, 0, 1782, 6188
[65]	LWE-976	CCA	xc7a35t	167	45455	1, 16, 1985, 7209
[44]	LWE-976	CCA	xczu9eg	402	760	32, 34, 1190, 7087 (*)
[44]	LWE-1344	CCA	xczu9eg	417	1328	32, 35, 1215, 7015 (*)
[44]	MLWR-512	CCA	xczu9eg	322	43	256, 7, 1989, 12343 (*)
[44]	MLWR-768	CCA	xczu9eg	322	49	256, 7, 1993, 12566 (*)
[87]	MLWR-768	CCA	xc7z020	125	4147	28, 4, n.a., 7400 (*)
[44]	MLWR-1024	CCA	xczu9eg	322	50	256, 7, 2341, 12555 (*)

compared to [105]. While their FPGAs are different (Virtex-6 v.s. Kintex-7), it seems that the NTT is more efficient than schoolbook multiplication.

Differences between the FPGA used may account for some of the surprising performance results in the table. The RLWR-1170 results by [44] were obtained for a high-end Zynq UltraScale+ FPGA. Their computation time is more than 30 times faster than the RLWR-1018 result by [5] using a Cyclone-V FPGA.

Since RLWR based cryptosystems often choose power of 2 moduli, their implementation involves schoolbook or Karatsuba style polynomial multiplication. Most RLWE implementations on the other hand, use the NTT. The computation of the NTT and its inverse is the bottleneck during both encryption and decryption. Several optimizations have been proposed to accelerate this computation. The multiplication by the powers of the $2n$-th root of unity can be merged with the twiddle factors in the first stage or the scaling multiplication by $n^{-1} \bmod q$ [105]. Instead of precomputing n^{-1} and the powers of ϕ^{-1}, the values of $n^{-1}\phi^{-i}$ for $0 \leq i < n$ can be precomputed directly. This saves one multiplication per coefficient. A similar result merging the NWC with the final stage of the inverse NTT was described by [98].

One technicality that comes with the Cooley-Tukey algorithm, is the so-called bit reversal. The output vector of the Decimation-In-Frequency (DIF) variant of the transform is permuted. The output coefficients are arranged in bit reversed order. For $n = 8$ for instance, the coefficient on position $1 = (001)_2$ can be found on index $(100)_2 = 4$. After the transformation, all coefficients have to be rearranged by reversing the bits of the indices. The Decimation-In-Time (DIT) transformation takes an input vector whose coefficients are in bit reversed order, and returns an output vector in the correct order. By making clever use of the DIT and DIF transforms, [98] shows that the bit reversal can be avoided. The DIF transform is used for the forward transformation, while the DIT transform is used for the inverse. The bit-reversal resulting from the DIF forward transformation is thus automatically undone by the inverse NTT. All the operations in the NTT domain are computed on the bit-reversed coefficient vectors. The public and private keys are therefore stored in bit-reversed order in the NTT domain. To limit the amount of modular reductions during the NTT, [79] allows variables to grow slightly larger than q.

Even with these optimizations, the NTT is still a very costly operation. To reduce the number of NTTs to be computed, the public and private keys can be stored in the NTT domain. The ciphertext part $\mathbf{c}_1$ must also be sent in the NTT domain. During the encryption, 2 forward NTTs and 1 inverse NTT have to be computed and during the decryption only 1 inverse NTT is needed.

LWE involves the storage of large matrices, which may be problematic on constraint devices. A simple trick is often used to avoid dealing with the largest matrix, the public key part $\mathbf{A}$. This matrix contains pseudo-random integers, generated using some deterministic pseudo-random number generator (PRNG). A PRNG takes as input some random value, a *seed*, and returns any number of cryptographically secure pseudo-random random numbers. If only the seed is stored in memory, and not the complete matrix generated by it, an important amount of space can be saved. The public key then consists of the seed and the smaller matrix $\mathbf{b}$. To use the public key for encryption, the coefficients of the matrix $\mathbf{A}$ are generated *on the fly*, concurrently with the coefficients of the error matrix $\mathbf{e}_1$. The coefficients of $\mathbf{A}$ are immmediately used in the multiplication with the coefficients of $\mathbf{e}_1$, and discarded afterwards. The encryption algorithm can thus be computed without storing the matrix $\mathbf{A}$. The same technique is proposed for the matrix or polynomial $\mathbf{a}$ in Kyber and NewHope respectively. LWE implementation results by [63] and [44] show that memory utilization is still an important issue. The ciphertext matrix $\mathbf{c}_1$ and other matrices that appear in the algorithms are, while much smaller than $\mathbf{A}$, still very large compared to the polynomials in RLWE. The computation time is also very high compared to RLWE.

The memory utilization results for MLWR in Table 3.2 show a compromise between LWE and RLWE, as one could expect by looking at the encryption algorithms. The large amount of DSPs used by [44] is a result of a high level of parallelization. They do not benefit from fast polynomial arithmetic using the NTT. To compensate for this and decrease the computation time, DSPs are added for parallel computation.

3.6 Side-Channel Attacks

In the event that an attacker has physical access to a device that is computing a cryptographic algorithm, *Side Channel Analysis* (SCA) may be used to obtain information about the encrypted message or the secret key. The first attacks using SCA were reported by [73]. These *timing attacks* exploit the dependence of the computation time on the secret key. By carefully measuring this computation time, information about this key can be obtained. *Simple Power Analysis* (SPA) [71] exploits the leakage of information caused by the power consumption variations of a device. Easy targets for SPA include the Square-and-Multiply algorithm for modular exponentation and the Double-and-Add algorithm for point addition on ECC [72][41]. The computations in these algorithms at instant $t = t_i$ depend directly on the i-th bit of the secret key. The Diffie-Hellman key exchange, RSA and ECC using these algorithms are particularly vulnerable to SPA attacks.

The power consumption at each precise instant can be modelled as a linear function of the number of non-zero bits of the values in the registers. This is called the Hamming Weight (HW) model, as the HW of the data in the registers determines the estimated power consumption. The Hamming Distance model [30] on the other hand, assumes that the power consumption is a linear function of the number of bits flipped at each instant. *Differential Power Analyis* (DPA) [71] can use either model to compare observed power traces with estimated power consumptions.

In [71] it is described how to use DPA to extract the keys from an implementation of the DES cipher. The trick is essentially to find an instruction in the algorithm that uses only a small part of the secret key dependent state, called *subkey*. The DES cipher uses Sboxes which take six key-dependent input bits and return four bits. Using the HW model, the power consumption of the Sbox can be predicted for all of the 2^6 possible inputs (called *subkey guesses*). Statistical tools are used to compare a number of observed power traces to the predicted consumptions. It can then be determined with some probability that one particular subkey guess is true (i.e. the guess corresponds to the true subkey value), while all other subkey guesses are false. The whole process is repeated for the other subkeys. This divide-and-conquer strategy eventually allows to recover the complete secret key.

Correlation Power Analysis [30][84] is a variant of DPA in which the Pearson correlation coefficient is used to compare between the observed and the predicted power traces. In order to perform a correlation power attack, a large number of random inputs $c_0, \ldots, c_N$ to the target device is generated. For each of these inputs c_i, and assuming some subkey guess s_0, the power consumption $P_{s_0}(c_i)$ of the device at a specific instance $t = t_0$ is estimated using a prediction model, yielding a vector of predicted power values:

$$\vec{P}_{s_0}^{(t)} = \left(P_{s_0}^{(t)}(c_0), P_{s_0}^{(t)}(c_1), \ldots, P_{s_0}^{(t)}(c_N) \right). \tag{3.13}$$

This computation is repeated for each possible subkey guess s_i, resulting in a number of vectors $\vec{P}_{s_0}^{(t)}, \vec{P}_{s_1}^{(t)}, \ldots$ of predicted values for each possible subkey guess s_i. For each of the random inputs, the actual power consumption of the device is measured at instance $t = t_0$, yielding a vector of observed values:

$$\vec{O}^{(t)} = \left(O^{(t)}(c_0), O^{(t)}(c_1), \ldots, O^{(t)}(c_N) \right). \tag{3.14}$$

The correlation between $\vec{O}^{(t)}$ and $\vec{P}_{s_i}^{(t)}$ is computed for each s_i. If the prediction model is accurate, then the maximum correlation is obtained for the subkey guess s_j that corresponds to the actual value of the subkey. The attacker repeats the process for the unknown remainder of the key.

DPA and correlation attacks require many traces in order to find a single subkey. The attacker is only interested in one particular time point of the trace, during which the targeted operation is executed. *Horizontal Correlation Power Analysis* [38] exploits the fact that a single subkey value may impact the value of many variables in the algorithm. In this case there are many points of interest $t_0, t_1, \ldots, t_N$ on a single trace. The observation vector from 3.14 is then taken from a number of points on a single trace instead, computing on the same fixed input $c = c_0$:

$$\vec{O} = \left(O^{(t_0)}, O^{(t_1)}, \ldots, O^{(t_N)} \right). \tag{3.15}$$

Similarly, the prediction vectors are computed for the fixed input $c = c_0$ and for all possible subkey guesses s_i:

$$\vec{P}_{s_i} = \left(P_{s_i}^{(t_0)}, P_{s_i}^{(t_1)}, \ldots, P_{s_i}^{(t_N)} \right) \tag{3.16}$$

The statistical analysis using the correlation coefficient remains unchanged. Correlation Power analysis using many traces for each subkey is sometimes referred to as *vertical* attack, as opposed to the horizontal variant.

In some situations, the estimated power traces from the Hamming difference model might not be accurate enough to succesfully attack a device. *Template attacks* [35] assume a more powerful attacker. Before the attack, a profiling phase allows to pre-record a large number of real power traces for all possible subkeys. In order to do this, the attacker must have a second device, similar to the target device, on which (part of) the same algorithm is implemented and on which the attacker can control the secret key.

Fault attacks [26] are active attacks in which the attacker perturbs the computing device in order to cause faults in the computation. The fault is typically caused by underpowering or overpowering the device, perturbing it with electromagnetic radiation, heating or cooling it or running it on too high a frequency (*overclocking*). Analysis of the erroneous output may, depending on the algorithm, reveil information about the secret key. One classic example is the Bellcore attack [26][67][8] targeting the RSA-CRT signature algorithm. This algorithm uses the Chinese Remainder Theorem (CRT) to compute an RSA signature in two parts. By injecting a fault into the computation of one of these parts, the factorization of the public key can be deduced from the faulty signature.

3.6.1 SCAs on Lattice Cryptography and Countermeasures

On error sampling

Like any unprotected implementation of cryptographic primitives, lattice-based cryptosystems are not resistant against SCAs. Many signature schemes and some encryption schemes rely on the ablility to sample from a discrete Gaussian distribution. Vulnerabilities in Gaussian sampling of the signature scheme BLISS [49] were used in cache based attacks by [32][93]. Several attacks on the same scheme have been discussed by [53]. Various leakages are caused by the rejection sampling in the signing algorithm. The Gaussian sampling algorithm used in the implementations is not constant time, making it vulnerable to timing attacks. A countermeasure proposed by [106] consists of applying a random permutation to the vector of Gaussian samples. In the case that the vector is used as an error vector, this random permutation does not impact the correctness of the scheme. There have also been made efforts to make the sampling algorithm run in constant time [69][70], but their propositions are less efficient and need more pseudorandom bits per sample than other sampling methods. The difficulties encountered with constant time Gaussian sampling has motivated some schemes to use noise distributions that are easier to sample, such as the binomial distribution (Kyber [27], NewHope [4]) or the uniform distribution (ring-Tesla [3], Dilithium [50], Saber [45]).

On modular and polynomial arithmetic

SPA and correlation power attacks can be used to target the polynomial multiplication. Sparse multiplication techniques in the implementation of BLISS can be attacked with one single trace using timing differences [53]. Multiple trace attacks like [101] use correlation power analysis to recover a single secret key coefficient successfully within 100 power traces. They propose propose a first-order masking scheme as a countermeasure. The masking scheme uses the linearity of the operations during the decryption. The non-linear decoding step requires a rather inefficient masked decoder, which has a heavy impact on the performance. Their second-order attack on the protected implementation is still successful but needs over 2000 traces.

A different solution uses the additive homomorphic properties of the encryption scheme [103]. The sum of two ciphertexts encrypts the sum of two plaintexts corresponding to each of the ciphertexts. To mask the decryption, first a random plaintext is encrypted. The resulting ciphertext is added to the input ciphertext. Decryption then yields the sum of the desired plaintext and the random one. The problem with this method is that it adds extra noise to the ciphertext, thereby increasing the decryption failure rate. Usually the parameters of encryption schemes are optimized in such a way that any additional noise may compromise the security of the scheme. Decryption failures can be exploited [51][52].

Other proposed countermeasures against DPA include *shifting* and *blinding* [106]. These protections consist of multiplying the input ciphertext and the secret key by some random scalar in $\mathbb{Z}_q$ or some power of x in $\mathcal{R}_q$ at the start of each decryption. The goal of this randomization of operations is to randomize the power traces. The blinding countermeasure was used in combination with masking in the CCA secure implementation by [91].

Power attacks on NTRU [7, 111, 66] typically target the schoolbook multiplication algorithm using vertical DPA attacks. Horizontal DPA attacks use information from multiple points on a single power trace. The horizontal DPA on NTRUPrime by [66] uses that fact that the same secret key coefficient is multiplied by many different known coefficients. A schoolbook multiplication between a ciphertext polynomial $\mathbf{c} = \sum_{i=0}^{n-1} c_i$ and a secret polynomial $\mathbf{s} = \sum_{j=0}^{n-1} s_j$ can be written as:

$$\begin{aligned}\mathbf{cs} &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} c_i s_j x^{i+j} \\ &= \sum_{j=0}^{n-1} s_j \sum_{i=0}^{n-1} c_i x^{i+j} \\ &= \sum_{j=0}^{n-1} \left(s_j c_0 x^j + s_j c_1 x^{j+1} + s_j c_2 x^{j+2} + \dots\right).\end{aligned}$$

Each secret coefficient s_j is multiplied by n known input coefficients. A single power trace with little noise could therefore provide sufficient information to recover all the secret key coefficients. A similar attack [9] applies to schoolbook multiplication when used in RLWE schemes. The attack can also be adapted to target the matrix multiplication in the standard LWE scheme Frodo, as described in the same paper. During a multiplication between one known and one secret matrix, each secret coefficient is used in many different integer multiplications. A multiplication of the ciphertext matrix $\mathbf{C}$ with a secret key vector $\mathbf{s}$ is computed as follows:

$$\mathbf{Cs} = \begin{pmatrix} c_{00} & c_{01} & c_{02} & \cdots \\ c_{10} & c_{11} & c_{12} & \cdots \\ c_{20} & c_{21} & c_{22} & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ c_{(n-1)0} & c_{(n-1)1} & c_{(n-1)2} & \cdots \end{pmatrix} \times \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_{n-1} \end{pmatrix}$$

$$= s_0 \begin{pmatrix} c_{00} \\ c_{10} \\ \vdots \\ c_{(n-1)0} \end{pmatrix} + s_1 \begin{pmatrix} c_{01} \\ c_{11} \\ \vdots \\ c_{(n-1)1} \end{pmatrix} + \ldots$$

$$= \begin{pmatrix} s_0 c_{00} \\ s_0 c_{10} \\ \vdots \\ s_0 c_{(n-1)0} \end{pmatrix} + \begin{pmatrix} s_1 c_{01} \\ s_1 c_{11} \\ \vdots \\ s_1 c_{(n-1)1} \end{pmatrix} + \ldots$$

Each secret key s_i is used in n modular multiplications. The amount of information from a single trace is sufficient to recover the complete secret matrix. A similar approach is used by [29] to attack Frodo.

Another attack angle is provided by the modular reduction. The non constant time of the Barrett reduction algorithm inside the NTT is used by [99]. Their template attack requires a profiling phase during which 100 millions traces are used to create close to a million templates. Once this profiling phase is completed, the information from one single trace, combined with the information of the templates, suffices to recover the secret key. To counter their attack, they suggest constant time implementations of modular reduction, or the shuffling of operations inside the NTT. The template attack targeting a constant-time Kyber implementation by [94] targets the NTT in the encryption in order to recover the message. The (unknown) input to the NTT in the encryption function is taken from a small subset of $\mathbb{Z}_q^n$, as opposed to the decryption. This allows to reduce the number of templates needed to 213.

Fault attacks

In [33] the lattice-based signature scheme Dilithium is attacked using fault injection. The fault allows to obtain two different signatures for the same message, which can then be used to compute the secret key. The attack exploits the determinism in the randomness generation of the signing algorithm.

Randomization of the sampling is suggested as a countermeasure. In general, fault attacks on signature scheme can often be prevented by verifying the signature before returning it as output, or computing the signature twice. A range of fault attacks on lattice-based signature schemes are discussed by [24]. Their proposed fault attacks require the ability to fault the device in such a way that it skips a specific instruction, or changes the value in some registers to zero or to some random number. Loop-abort faults consist of forcing an early termination of some loop. This can be achieved by changing the value of the loop counter. In [54] it is described how to deploy this type of attack against signature schemes and encryption schemes such as BLISS and NewHope respectively. Most lattice-based schemes use the sampling of some noise vector that is added in order to hide key-dependent variables. If this noise sampling were to be terminated in an early stage, then some of the key-dependent variables would be visible in plain sight. The remainder of the secret key can then be recovered by solving the underlying lattice problems, only now on a lattice of a significantly lower dimension. Algorithm 11 shows an example procedure typically found in signature and encryption algorithms. The output is a secret vector or polynomial with normally distributed coefficients. By faulting the loop counter to cause an early termination of the loop, the output vector may have very few non-zero coefficients.

Algorithm 11 Sampling randomness

```
function SAMPLE
    $\mathbf{e} \leftarrow (0, 0, 0, \ldots, 0)$
    for $i \leftarrow 0$ to $n - 1$ do
        $e_i \xleftarrow{\$} \mathcal{N}_\sigma(\mathbb{Z}_q)$
    return e
```

Similar methods are used to attack the Frodo key exchange. Countermeasures against this type of attack include implementing double loop counters, or verifying that the loop has been exited correctly.

Chapter 4

Implementation Environment

4.1 Introduction

This chapter introduces the methodology and the tools used in our implementations. After a brief introduction to FPGAs and HLS, we show how HLS can be efficiently used to implement cryptographic applications on FPGA. Section 4.2 discusses the implementation of finite-field arithmetic, which is at the core of many PKC primitives. The modular reduction operator native to C language yields poorly performing implementations when used in HLS. We describe how to improve both area utilization and computation time by using customized algorithms. We also exploit the arithmetic properties of specific forms of moduli to reduce area utilization. In section 4.3 we discuss the behaviour of Xilinx Vivado HLS tool, using the schoolbook algorithm for polynomial multiplication as an example. By rewriting the algorithm and using a specific set of *directives*, the implementation results from HLS can be significantly improved.

4.1.1 FPGAs

Field Programmable Gate Arrays (FPGA) are devices that consist of a two dimensional array of programmable logic (*Configurable Logic Blocks*, CLB), whose elements can be interconnected. Programming an FPGA consists of programming (a part of) the logic elements on the array and the interconnect in a certain way. FPGAs were first introduced by Xilinx [31]. The programmable array elements in their FPGAs are called *slices*. The main components of a slice are *look-up tables* (LUT) and *flip-flops*. The Xilinx 7 series LUT for example, takes up to six input bits and returns one output bit. By setting the entries of the table, it can be configured as any boolean function that takes six inputs. In order to construct more complicated functions, multiple LUTs, each one programmed as some boolean funcion, can be interconnected. Such constructions can be used to implement parts of finite-field operations.

Xilinx FPGAs also contain dedicated elements for integer operations, called *digital signal processing*

(DSP) blocks. The DSP48E1 block from the Xilinx 7 series [112] can be used to multiply a 25-bit signed integer with an 18-bit signed integer. Additionally, the slice contains logic that computes addition or subtraction with a third or fourth input before or after the multiplication. Given four inputs a, b, c and d, the DSP block computes $(a + b) \times c + d$. The registers on the DSP can be used to store the result and accumulate new results to it. Complete multiplication and accumulation loops computing functions of the form $\sum_{i=0}^{n}(a_i + b_i) \times c_i$, can be implemented on a single DSP block. Alternatively, one may choose to compute $\sum_{i=0}^{n/2-1}(a_i + b_i) \times c_i$ on a first DSP block, and $\sum_{i=n/2}^{n}(a_i + b_i) \times c_i$ on a second one. The final result can be obtained by adding the results of the two partial summations together. The length of the loop, which determines the number of clock cycles (CC) needed to perform the summation, is divided by 2. An implementation splitting the sum over 2 DSP blocks (computing simultaneously) can thus compute the final results twice as fast as an implementation using only 1 DSP block. This technique of accelerating the computation is referred to as *parallelism*, as the two DSP blocks operate in parallel.

Another essential building element in FPGA implementations is the *multiplexer* (MUX). It allows to select one output from two inputs. A third input, the *selection bit*, is used to choose the desired output. The `If/Else` statements in algorithmic descriptions are typically implemented using multiplexers. The instructions for both `If` and `Else` are implemented by some logic, and both results are sent to a multiplexer. This multiplexer selects the right output by using the outcome of the `If` condition (`True/False`) as a selection bit. Multiple multiplexers can be interconnected in order to select between more than two inputs.

LUTs, DPS blocks and multiplexers can be used to implement the logic and arithmetic parts of an algorithm. In order to compute a sequential algorithm, some intermediate results may need to be stored for later use. The storage can be done using the hardwired memory blocks available on the FPGA. These memory units are called *block RAMs* (BRAM). Intermediate values can be written to or read from a BRAM. The amount of information that can be stored in a single BRAM is limited. The Xilinx 7 series FPGAs have BRAMs that can store up to 36 Kbits of data [114].

To design an efficient FPGA architecture that implements some algorithm, the use of resources must be optimized. The number of slices, DSPs and BRAMs available on the FPGA depends on the specific device, and is correlated with its price. The smallest Xilinx' Artix 7 part XC7A12T for instance, has only 2,000 slices, 720 Kb of BRAM and 40 DSPs. The more expensive device XC7A200T of the same Artix 7 series has over 33,000 slices, 13 Mb of BRAM and 740 DSP blocks. Even more expensive devices such as the Kintex 7 part XC7K480T contain almost 2000 DSPs. By reducing the area utilization of an FPGA architecture, the implementation may fit on a much cheaper FPGA. It may therefore be worthwhile to optimize the area efficiency of the architecture. This can be done by reusing area resources for multiple instances of similar computations in the implemented algorithm. However, this may results in slower implementations, since it reduces parallelism. In practice, there is often a trade-off between the area utilization and the computation time of the implementation.

4.1.2 High Level Synthesis

FPGA implementations can be created using a Hardware Description Language (HDL) such as VHDL and Verilog. Programming in these languages can be a cumbersome process. This process can be avoided by describing the algorithm in C language and then implementing it on FPGA using High Level Synthesis (HLS) [113][42]. HLS is a tool that takes as input an algorithm in C and generates an FPGA implementation that implements the algorithm. HLS can be used as a shortcut to obtain FPGA implementations relatively quickly compared to VHDL/Verilog methods. This enables the ability to explore a hardware design space at a reduced cost. HLS can therefore be a useful tool in the development process of hardware accelerators.

4.1.3 HLS and Cryptography

Traditionally, HLS is mostly used in the domain of digital signal processing [83]. HLS tools are not specifically designed for cryptographic purposes, and their use is not widespread among cryptographers. The potential of HLS however, including a reduced and simplified development process, makes a case for its use in cryptography. In [61] the performances of FPGA implementations of the AES cipher, generated by Xilinx' Vivado HLS, are compared to those of handwritten Register-Transfer Level (RTL) implementations using VHDL of the same algorithm. The area utilization of their HLS generated implementations is very similar to that of their RTL implementations. In terms of throughput, however, the RTL implementation is better. Depending on the FPGA, the throughput is reduced by 23 to 47 percent when using HLS.

Xilinx' Vivado HLS was also used by [60] to evaluate the performances of 16 competing algorithms in the CAESAR contest for authenticated ciphers. Using the HLS implementation results, they create a ranking, allowing to compare between the throughputs and throughput to area ratios of the implemented algorithms. Their work shows that benchmarking candidates using HLS can be particularly helpful in cryptographic contests. This method was used by [62] to compare 5 hash functions in the SHA-3 contest for secure hashing algorithms. The more recent work by [44] takes a similar approach using a software/hardware codesign to compare 3 lattice-based KEMs (Frodo, Round5 and Saber) in the NIST post quantum competition.

4.2 Finite-Field Arithmetic using HLS

This section is based on our joint work [48] with Libey Djath. In this section we discuss the implementation of finite-field arithmetic on FPGA usign HLS. Cryptographic applications often require computations in finite fields of the form $\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z}$ for some fixed prime number q. We implement operations typically encountered in cryptographic applications. The computation of $\sum_{i=1}^{n} x_i \times y_i \bmod q$ for some input vectors $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ of some vector length n, where the

x_i, y_i are w-bit values for instance, is a core operation found in matrix multiplication algorithms, or schoolbook polynomial multiplication.

In Vivado implementations, modular reduction, represented by the % operator in C language, is computed by implementing a general modular reduction algorithm, most likely a Euclidean division, that could work for any modulus q. The fact that q is constant, is not exploited by Vivado. Reduction algorithms for fixed moduli, such as Barrett [17] and Montgomery [88] algorithms, may yield better performances when implemented using HLS.

Moreover, in some cryptosystems the modulus q is an integer whose binary decomposition has some specific structure that simplifies modular arithmetic. The best example is $q = 2^w$ for some integer w. Any integer $x > q$ can be written as $x_0 + 2^w x_1$ for some $x_0 < 2^w$. Then $x \bmod 2^w = x_0 + 2^w x_1 \bmod 2^w \equiv x_0$, that is, the modular reduction for $q = 2^w$ can be computed by taking the w least significant bits and discarding all the other bits. A specific structure in the binary decomposition of the modulus should therefore not be overlooked when implementing modular arithmetic. In this section we consider examples of non-trivial prime moduli with some specific binary decomposition. We define the following:

- MQ: moduli without any particular structure in their binary decomposition.
- MSC: moduli of the form $2^w - 2^l + 1$ for integers w and l such that $w > l > 0$.
- MSR: moduli that are close to some power of 2, which can be written as $2^w \pm c$ for $c < 2^{w/2}$.

Moduli of the form MSC include $7681 = 2^{13} - 2^9 + 1$, used in Kyber key exchange mechanism. The algorithm that computes modular reduction for MSC can be computed using only add and bit-wise shift operations. We describe a fast reduction algorithm in the style of [109] for MSC moduli. Let $x < 2^{2w+d}$ for some integer d, then there exist $0 \leq x_0 < 2^w$ and $0 \leq x_1 < 2^{w+d}$ such that $x = 2^w x_1 + x_0$. Using the fact that $2^w \equiv 2^l - 1 \bmod q$, one has:

$$x \equiv (2^l - 1)x_1 + x_0 = 2^l x_1 - x_1 + x_0. \tag{4.1}$$

Let $x^{(1)}$ denote the value on the right hand side of the equation above. Then $x^{(1)} < 2^{l+w+d} + 2^w$ and

$$x^{(1)} = 2^w x_1^{(1)} + x_0^{(1)} \tag{4.2}$$

for some $x_1^{(1)} \leq 2^{l+d}$ and $x_0^{(1)} < 2^w$. Again substitute 2^w by $2^l - 1$ and obtain:

$$x^{(1)} = (2^l - 1)x_1^{(1)} + x_0^{(1)} = 2^l x_1^{(1)} - x_1^{(1)} + x_0^{(1)} =: x^{(2)}, \tag{4.3}$$

where $x^{(2)} \leq 2^{2l+d} + 2^w$. Repeating this procedure, it follows that

$$x \bmod q \equiv x^{(i)} < 2^{2w+d-i(w-l)} + 2^w. \tag{4.4}$$

This means that for $i \geq \frac{w+d}{w-l}$ one has that $x^{(i)} < 2^{w+1}$. Algorithm 12 can be computed using only bit-wise shifts and additions/subtractions. Optimizations similar to those in [110] may be used to remove lines 6 and 7 of the algorithm.

Algorithm 12 Compute $x \bmod q$ for MSC ($q = 2^w - 2^l + 1$)

```
1: function $\text{RED}_{w,l}(x)$
2:     for $i \leftarrow 1$ to $\left\lceil \frac{w+d}{w-l} \right\rceil$ do
3:         $x_0 \leftarrow x \bmod 2^w$
4:         $x_1 \leftarrow \left\lfloor \frac{x}{2^w} \right\rfloor$
5:         $x \leftarrow 2^l x_1 + x_0 - x_1$
6:     if $x \geq q$ then
7:         $x \leftarrow x - q$
8:     return $x$
```

Let for example $q = 2^{13} - 2^9 + 1 = 7681$ and $x = 1234567$. To compute $x \bmod q$, we first compute x_0 and x_1 :

$$x_0 = 1234567 \bmod 2^{13} \equiv 5767$$

and

$$x_1 = \left\lfloor \frac{1234567}{2^{13}} \right\rfloor = 150.$$

Then for the next iteration we have :

$$x^{(1)} = 2^9 \cdot 150 + 5767 + 150 = 82417.$$

We continue and compute $x_0^{(1)}$ and $x_1^{(1)}$:

$$x_0^{(1)} = 82417 \bmod 2^{13} = 497$$

and

$$x_1^{(1)} = \left\lfloor \frac{82417}{2^{13}} \right\rfloor = 10.$$

Then

$$x^{(2)} = 2^9 \cdot 10 + 497 - 10 = 5607 < q.$$

Therefore $1234567 \bmod 7681 = 5607$.

Moduli MSR (of the form $2^w \pm c$) can also benefit from the particular structure of their binary decomposition. Let $q = 2^w + c$ for some small c and let $q \leq x < q^2$ such that $x = x_0 + 2^w x_1$ for some $x_0 < 2^w$. Then

$$\begin{aligned} x \bmod q &= x_0 + 2^w x_1 \bmod q \\ &\equiv x_0 + (-c) \cdot x_1 \\ &= x_0 - cx_1 \end{aligned}$$

The value $x_0 - cx_1$ is equivalent to $x \bmod q$, and can be written on $w + \log_2(c) < \frac{3w}{2}$ bits. By repeating this procedure, smaller values in the same equivalence class can be found. This yields an algorithm for finding $x \bmod q$. We also implement Barrett [17] and Montgomery [88] reduction for moduli MQ (without any particular structure). The source code of our implementation of the Barrett algorithm is shown in figure 4.2 (the notations used in the figure are explained below). The reduction algorithm is to be used in computations of the form $(\sum_{i=1}^{n} x_i \times y_i) \bmod q$. The code uses the arbitrary precision integer library, available for HLS. This library allows to define integer types of arbitrary precision, which optimizes the resources allocated to perform certain operations. A multiplication of two integers of the 32-bit type `int` is implemented using multiple DSP blocks, since one DSP block only allows to compute multiplications of an 18-bit signed integer with a 25-bit signed integer. If the `int` is used for values that fit within this range, than it would not be necessary to allocate multiple DSPs to compute the multiplication. By defining an integer type of 18 bits for example, this overspending of DSPs can be avoided. The HLS tool will recognize that the types fit into one single DSP block, and therefore implement only one DSP. The following types are defined:

- `word` : unsigned integer of w bits;
- `dword` : unsigned integer of $2w$ bits;
- `sumword` : unsigned integer of $w + \lfloor \log_2(n) \rfloor + 1$ bits, where n is the size of the summation.
- `sumdword` : unsigned integer of $2w + \lfloor \log_2(n) \rfloor + 1$ bits.
- `counter` : unsigned integer of $\lfloor \log_2(n) \rfloor + 1$ bits.
- `signword` : signed integer of $w + 1$ bits.

4.2.1 Implementation results

We use Vivado HLS version 2017.4 to implement the top level function (see figure 4.2) computing $\left(\sum_{i=1}^{20} x_i \times y_i\right) \bmod q$ on a Xilinx Artix-7 xc7a15 FPGA. Results for moduli of $13, 17, 23$ and 30 bits are shown in figure 4.3. It can be clearly seen in the graphs, that the implementation of the `%` operator is the slowest. Moreover, it uses far more slices than any of the other implemented algorithms. For instance, the `%` implementation for $w = 17$ has a computation time of between 2 to 2.5 times slower

```
#include "parameters.h"

word barrett(sumdword x)
    {
        sumword x1 = SUM_W(x >> width);
        sumword q = SUM_W((RSW(x1) * RSW(R_const)) >> (shift - width));
        word x0 = W(x);
        counter c = 0;
        if (x0 > M) c = 2;
        else if (x0 != 0) c = 1;
        q = q + c;
        sumdword z = SUM_DW(q) * SUM_DW(m);
        signword res = x - z;
        if (res < 0) res = res + M;
        if (res < 0) res = res + M;
        return W(res);
    }
```

Figure 4.1: Source code for Barrett reduction algorithm.

```
#include "parameters.h"
#include "arithmod.h"

word m2_rsf(word A[N], word B[N])
    {
    sumdword res=0;
    acc: for(counter i=0; i<N; i++)
        res += DW(A[i]) * DW(B[i]);
    return barrett(res);
    }
```

Figure 4.2: Source code for the implemented algorithm using the Barrett function.

algorithm and method	area		time (ns, cycles)			area×time	
	slices	DSP	period	cycles	TM	DSP×TM	slices×TM
Montgomery RIS	194	12	2.6	60	156	1872	30264
Montgomery RSF	149	7	2.6	64	167	1165	24794
Barrett RIS	259	12	2.8	53	149	1781	38436
Barrett RSF	218	10	2.7	52	141	1404	30608
MSC RIS	403	4	2.7	55	149	594	59846
MSC RSF	261	4	2.7	47	127	508	33121
MSR RIS	146	8	2.6	31	81	645	11768
MSR RSF	167	6	3.2	46	148	884	24583

Table 4.1: Different methods to compute $\sum_{i=1}^{20} x_i \times y_i \bmod q$ for $w = 23$.

than the others, while using 3 to 6 times more slices. In contrast to the Barrett and Montgomery implementations, however, the modular reduction by % does not use any DSPs. The 2 DSPs shown are used in the multiplication and accumulation loop. When looking at the product of the number of DSPs and the computation time (DSP × TM), a slightly different picture emerges. In this aspect, the % operator is up to 20 percent better than the Barrett implementation. This results from the Barrett algorithm's reliance on integer multiplication, and therefore an increased DSP utilization. The Montgomery implementation on the other hand, still has a better computation time/DSP utilization trade-off than %, although it has less of an advantage than when considering the computation time results only. Taking into account the trade-off between computation time and the utilization of slices (slices × TM), it becomes decisively clear that the % operator is not a viable option for any efficient finite-field arithmetic implementation. It shows 5 to 12 times worse performance in the slices × TM column, compared to the other algorithms. Any application that requires modular arithmetic for a fixed modulus, should not use the % operator in HLS. This confirms that a specialized finite-field arithmetic library is needed when using HLS for cryptographic purposes.

Figure 4.3 also shows the advantage that specialized algorithms for moduli of types MSC and MSR have over general algorithms (Barrett and Montgomery) for MQ. MSC does not require the use of DSP blocks, and therefore has the best DSP utilization/computation time trade-off. MSR benefits from similar properties, although for larger w additional DSP blocks are needed. The MSR algorithm computes multiplications with a constant c (determined by de modulus $q = 2^w \pm c$) where $c < 2^{w/2}$. For $w = 13$, we have that $c < 2^7$ and the HLS tool implements multiplication by 7-bit constants without using DSP blocks. For larger w the constant c is too large to be implemented using this alternative method, and therefore DSPs are instantiated. With respect to the slices/computation time trade-off however, MSR has the upper hand over MSC.

Even when choosing the most efficient modular reduction algorithm, other factors may impact the performance of the implementation. The operation that we implement consist of computing the modular reduction of a sum of integer multiplications. There are two different methods to implement

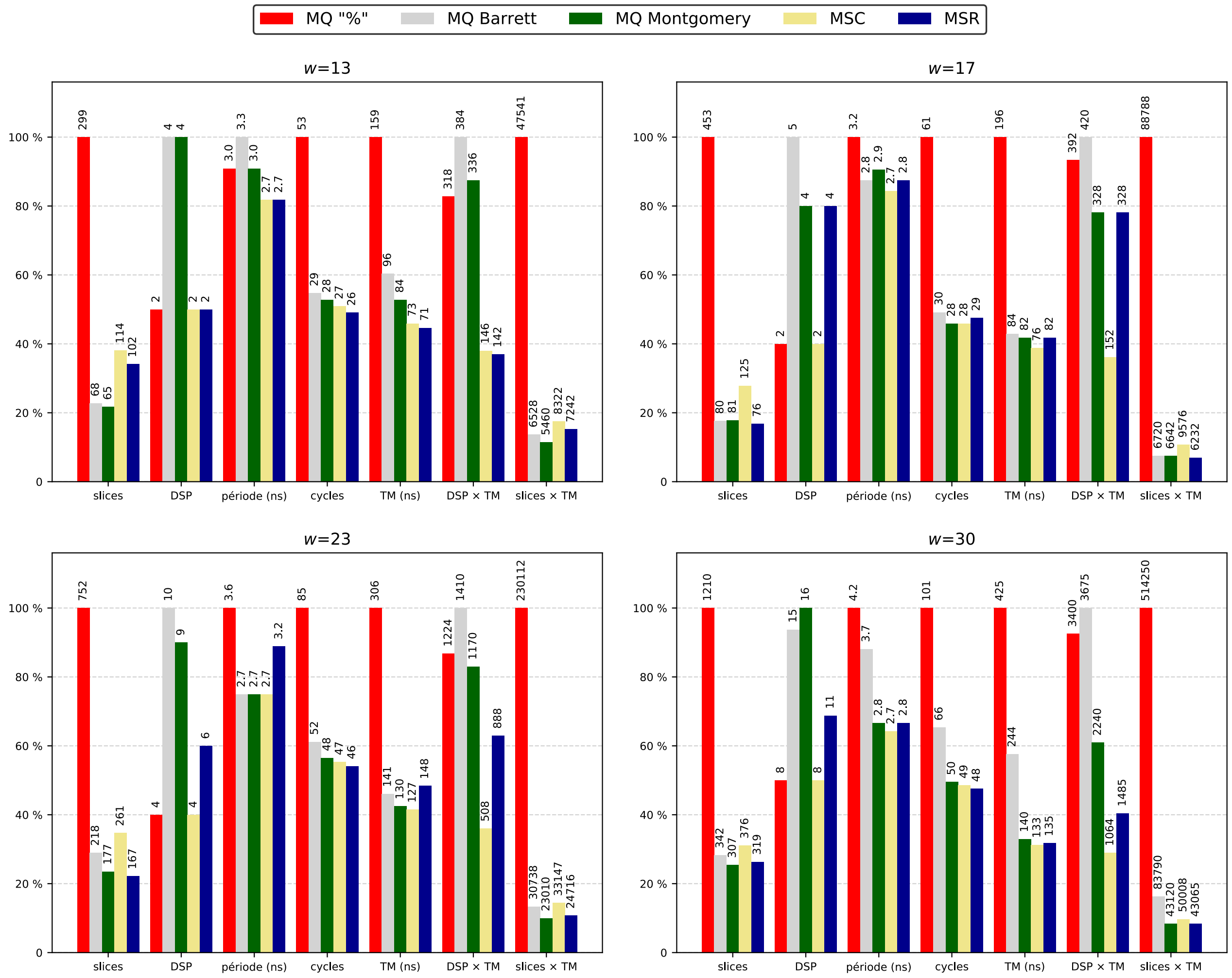


Figure 4.3: Comparison of different reduction algorithms for $w \in \{13, 17, 23, 30\}$ bits for the computation of $\left(\sum_{i=1}^{20} x_i \times y_i\right) \bmod q$.

the operation computing the reduction of a sum of products. We define the methods:

1. **RSF**: Compute the sum of products first, and one single modular reduction at the end:

$$\left(\sum_{i=1}^{n} x_i \times y_i\right) \bmod q$$

2. **RIS**: Compute the sum of reduced products and reduce the summation:

$$\left(\sum_{i=1}^{n} (x_i \times y_i \bmod q)\right) \bmod q$$

While one might expect the first method to be faster, this is not evident. The additions to be computed involve operands twice the size $w = \lfloor \log_2(q) \rfloor + 1$ bits of the vector elements $x_1, \ldots, x_n, y_1, \ldots, y_n$. The final modular reduction operates on an input of size $2w + \lfloor \log_2(n) \rfloor + 1$, which can be more costly to implement than the modular reduction of an integer of size $w + \lfloor \log_2(n) \rfloor + 1$, found in the second method. The extend to which one of these methods is better than the other, depends on the parameters q and n. We implement both algorithms for parameters $w \in \{13, 17, 23, 30\}$ and $n \in \{10, 20, 40, 100\}$. Table 4.1 shows the results of the different methods of computing the modular reduction of a sum of products. It can be seen in the table that the RIS method requires more DSP blocks for all of the implemented reduction algorithms. It also needs more slices (except for MSR). For Montgomery and MSR, the RIS method is faster than RSF. For Barrett and MSC, the opposite holds. The best area efficiency, measured by the DSP/computation time and the slices/computation time trade-offs, seems to be obtained for RSF. The MSR algorithm however, is an exception. In figure 4.4, the difference between RSF and RIS (using MSR algorithm) is shown for different vector lengths n. For all of the tested values of n, RIS has better trade-offs than RSF. Without surprise, the computation time and area utilization increases with n. Interestingly though, the cost of RSF seems to increase faster than RIS. In Table 4.1, the differences between RIS and RSF in the DSP×TM column range from 15% (MSC) to 40% (Montgomery). This shows that it is worthwhile, although not easy, to identify the right algorithm for a given application.

4.3 Schoolbook Algorithm for Polynomial Multiplication

In this section we describe how the choice of algorithm, implementation style and directives impact the quality of the implementation generated by the HLS tool. We use Vivado HLS v2017.4. We consider the schoolbook algorithm for polynomial multiplication in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ where $q = 7681$ and $n = 256$. Let $\mathbf{a}, \mathbf{b} \in \mathcal{R}_q$. The algorithm computes the coefficients c_k of the product polynomial

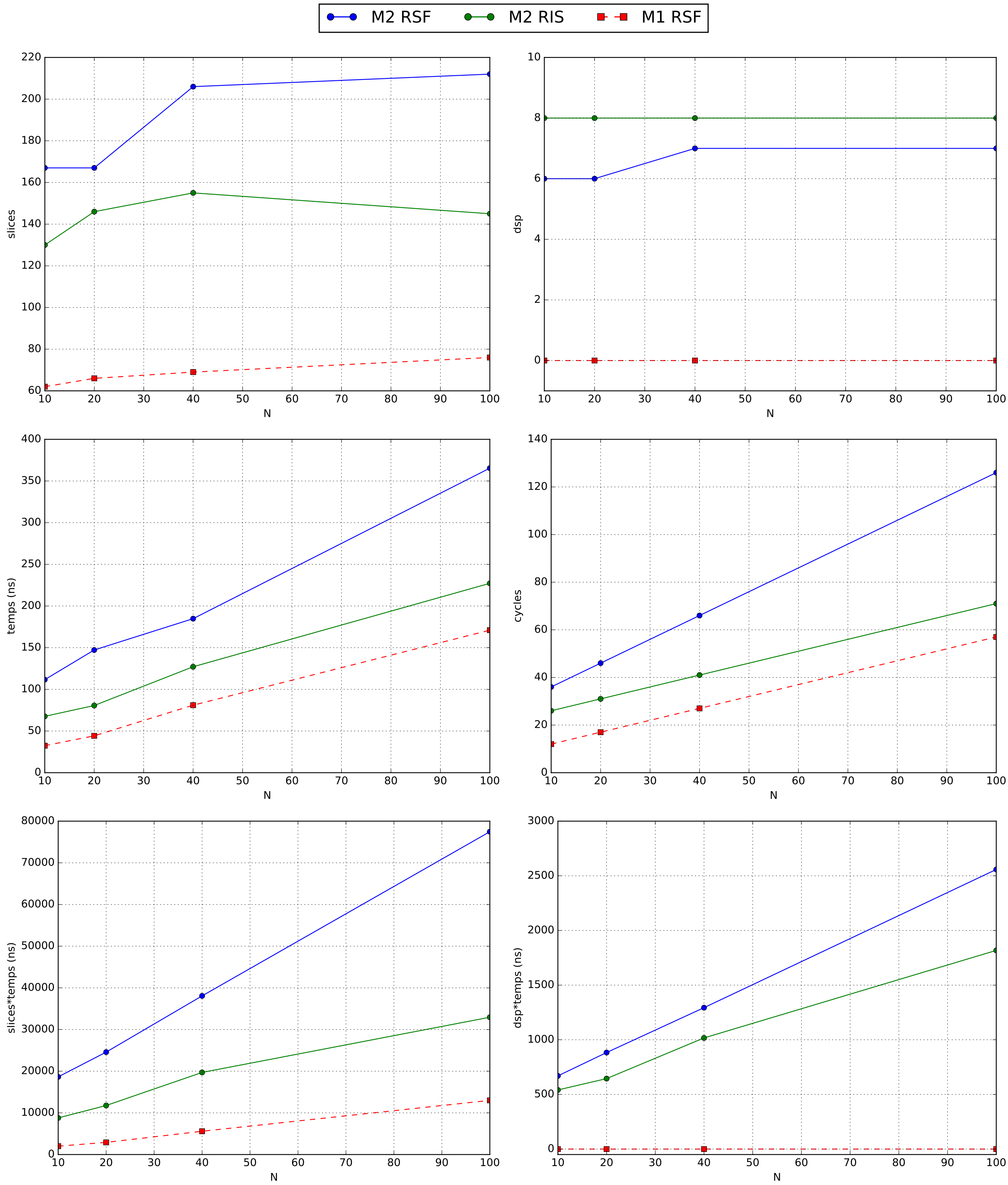


Figure 4.4: Impact of the vector length n on the computation of $(\sum_{i=1}^{n}(x_i \times y_i \bmod q)) \bmod q$ (RIS) and $(\sum_{i=1}^{n} x_i \times y_i) \bmod q$ (RSF) for $w = 23$, using the MSR. The red dotted line marked M1 RSF shows the results for implementations computing the modular reduction of a sum: $(\sum_{i=1}^{n} x_i) \bmod q$.

$$\mathbf{c} = \sum_{k=0}^{n-1} c_k x^k = \mathbf{a} \cdot \mathbf{b} \bmod (x^n + 1)$$

These coefficients can be expressed in terms of the coefficients of $\mathbf{a} = \sum_{i=0}^{n-1} a_i x^i$ and $\mathbf{b} = \sum_{i=0}^{n-1} b_i x^i$. Since $x^n \equiv -1 \bmod (x^n + 1)$, the computation consists of two summations for $0 \leq k < n$:

$$\begin{aligned} c_k &= \sum_{i+j=k} a_i b_j - \sum_{i+j=n+k} a_i b_j \\ &= \sum_{i=0}^{k} a_i b_{k-i} - \sum_{i=k+1}^{n-1} a_i b_{n+k-i} \end{aligned} \tag{4.5}$$

In order to minimize the number of write operations to the memory, we only consider algorithms in which each c_k is computed seperately. Once a coefficient c_k is completely computed, it is written to the memory. The most direct way of implementing the computation is shown in the source code in figure 4.5. The code uses custom integer types from the `ap_int` library. A 13 bit signed integer for example, exactly fits in the type `int13`. We define types `word`, `dword` and `sum_dword` as 13-bit, 18-bit and 26-bit signed integers respectively.

A Barrett reducer similar to the one from the previous section (implemented by the function `reduce`) is used for modular reduction. The coefficients of the polynomial **b** are small (5 bits) and are stored as constants in the memory, because in cryptographic applications the polynomial **b** is the secret key. The function therefore takes an input coefficient vector of **a** and returns the output coefficient vector of **c**. Note that the index bounds of the two inner loops depend on the index k of the coefficient c_k.

The variable loop bounds may prevent the HLS tool from optimizing a pipelined architecture. To transform the function `schoolbook0` into a function with constant loop bounds, we set all the lower bounds to 0 and the upper bounds to N. This will double the number of multiplications to be computed, but half of them are multiplications by zero. To avoid negative indices or indices out of range, we append zeroes to the left and right of the vector `b`. The resulting source code is shown in figure 4.6.

Before running the HLS tool, a number of choices have to be made. Firstly, the target device, an Artix-7 in our case, which is the FPGA for which the HLS tool will try to synthesize the code. Then a target clock period must be set. The HLS will try to optimize the generated implementation in such a way that the obtained clock period will be close to the target clock period. We synthesise the functions for target periods of $2, 4, 6$ and 8 ns. Optionally, a number of *directives* can be set. These directives provide additional information about the desired implementation. We compare the different levels of parallelism that can be obtained for the functions by applying the **unroll** directive

```
void schoolbook0(word a[N], word c_out[N]){
    counter k,i;
    sum_dword sum1, sum2;
    for(k=0; k<N; k++){
        sum1 = 0;
        sum2 = 0;
        for(i=0; i<=k; i++)
            sum1 += a[i]*b[k-i];
        for(i = k+1; i<N; i++)
            sum2 += a[i]*b[N+k-i];
        c_out[k] = reduce(sum1 - sum2);
    }
}
```

Figure 4.5: Straightforward implementation of the schoolbook polynomial multiplication.

```
void schoolbook1(word a[N], word c_out[N]){
    counter i,k;
    sum_dword sum1, sum2;
    for(k=0; k<N; k++){
        sum1 = 0;
        sum2 = 0;
        for(i=0; i<N; i++){
            sum1 += a[i]*b_extended[N+k-i];
            sum2 += a[i]*b_extended[2*N+k-i];
        }
        c_out[k] = reduce(sum1 - sum2);}
}
```

Figure 4.6: Implementation avoiding variable loop bounds.

(with factor 2 and 4) to the inner loops of the functions. Unrolling a loop with some factor f makes the HLS tool implement f instances of the required hardware to execute the loop. A loop of length n can then be computed in $\frac{n}{f}$ cycles by dividing the computations over all hardware instances. The directive **pipeline** is applied to the inner loops of all of the solutions, in order to pipeline the loops. This technique consists of implementing registers between computational units, allowing each unit to compute on a different input. Ideally, a loop of n iterations is computed in just over n cycles when perfectly pipelined. Arrays are by default implemented on a single BRAM. To split up an array over multiple BRAMs, the directive **array partition** can be set for a given factor. The array is then implemented in a given number of BRAMs. When unrolling a loop, multiple simulteneous accesses (memore reads) to the one single array are needed. Since one BRAM can only be accessed twice per clock cycle, we partition the arrays whenever the loops are unrolled.

The variables to be fixed for each implementation include the target period, the unroll factor and the algorithmic description. The resulting design space is too large to be explored manually, which is why we use a TCL script to iterate over all variables while generating the HLS implementations. The implementation results are shown in Table 4.2. The results are ordered by their computation time (latency × period). For a target period of 2 ns and without unrolling, the latency of the `schoolbook0` implementation is too high. In a smoothly pipelined implementation, the n^2 modular multiplications would be computed in about $n^2 = 65,536$ clock cycles. Instead, it takes twice as many (137,212) clock cycles. The variable loop bounds prevent pipeline optimization as expected. For a target clock period of 4, 6 or 8 ns (and without unrolling) we obtain implementations that compute the algorithm in around n^2 clock cycles indeed. Unrolling with factors 2 and 4 divide this number of cycles by almost 2 and 4 respectively, but the clock period exceeds the target. By unrolling the loop with factor 4 and relaxing the clock period requirement, we get surprising results. We obtain implementations running at clock period higher than 10 ns, taking around 20,000 clock cycles. These are the fastest results for `schoolbook0`, despite the high clock period. Similarly, for an unrolling factor 2 we obtain implementations whose clock period is much higher than the target (over 7 ns for target periods 4, 6 and 8 ns). The only implementations whose actual clock period is below the target, are given by those that do not use the unroll directive. They do not benefit from parallelism however, and are slower than the unrolled implementations.

The 7 fastest results in Table 4.2 are all implementations of `schoolbook1`, which justifies the rewriting of the algorithm. The obtained clock periods all are close to the targets. The most sequential implementation (without unrolling) executes the algorithm in around n^2 cycles for a target period of 2 ns. It does not meet the target clock period requirement however. This is probably caused by the fact that during each clock cycle, two multiplications have to be computed. The memory that stores $\mathbf{b}$ is accessed twice for different indices, slowing down the implementation. This problem is solved by adding BRAMs such that for unrolling factors 2 and 4, the performance of the implementation is greatly increased.

Table 4.2: All solutions for `schoolbook0` and `schoolbook1` (algorithms 0 and 1 in the first column respectively) and for all target clock periods. The solutions are ordered by computation time, from fastest to slowest.

algo.	unroll	target	obtained	latency	computation	area	
(0/1)	factor	per. (ns)	per. (ns)	(CC)	time (μ s)	DSP	LUT
1	4	2	1.87	39169	73	10	767
1	4	4	3.76	20737	78	10	812
1	4	6	4.63	19457	90	10	569
1	2	2	1.87	71169	133	6	467
1	2	4	3.76	36609	138	6	376
1	4	8	8.25	18689	154	10	560
1	2	6	4.63	35841	166	6	381
0	4	8	11.43	20033	229	10	1333
0	4	6	10.59	21825	231	10	1240
0	-	2	1.87	137212	257	4	488
0	2	2	3.52	72961	257	6	690
0	4	2	3.52	73985	260	10	1222
0	-	4	3.76	70653	266	4	437
0	2	6	7.06	37761	267	6	642
1	-	2	4.14	65556	271	4	551
0	2	4	7.06	38785	274	6	645
1	-	4	4.39	65551	288	4	543
1	2	8	8.32	34817	290	6	381
1	-	6	4.52	65548	296	4	543
0	-	6	4.52	69630	314	4	434
0	4	4	14.12	22337	315	10	1499
0	2	8	9.88	35713	353	6	483
0	-	8	6.35	68096	432	4	361
1	-	8	8.32	65543	545	4	511

```
void schoolbook1b(word a[N], word c_out[N]){
    counter k,i;
    sum_dword temp_out;
    for(k = 0; k<N; k++){
        temp_out = 0;
        for(i = 0; i<N; i++){
            temp_out = temp_out + a[i]* b_tilde[N - 1 + i - k];
        }
        c_out[k] = reduce(temp_out);
    }
}
```

Figure 4.7: The structure of the source code is simplified by rewriting the algorithm, using equation (4.6).

By rewriting the coefficient vector in a different order, we can improve the performance of `schoolbook1`. Let $\tilde{b}$ be the vector of length $2n$ defined by:

$$\tilde{b} = (b_{n-1}, b_{n-2}, \ldots, b_1, b_0, -b_{n-1}, -b_{n-2}, \ldots, -b_1, -b_0)\,,$$

that is:

$$\tilde{b}_i = b_{n-1-i}$$

and

$$\tilde{b}_{n+i} = -b_{n-1-i}$$

for $0 \leq i < n$. In other words, $\tilde{b}$ is the concatenation of the coefficient vector of $b(x)$ with its additive inverse. Then for $0 \leq k < n$ we have:

$$\begin{aligned}
\sum_{i=0}^{n-1} a_i \tilde{b}_{n-1+i-k} &= \sum_{i=0}^{k} a_i \tilde{b}_{n-1+i-k} + \sum_{i=k+1}^{n-1} a_i \tilde{b}_{n-1+i-k} \qquad (4.6)\\
&= \sum_{i=0}^{k} a_i b_{n-1-(n-1+i-k)} + \sum_{i=k+1}^{n-1} a_i \cdot (-b_{n-1-(-1+i-k)})\\
&= \sum_{i=0}^{k} a_i b_{k-i} - \sum_{i=k+1}^{n-1} a_i b_{n+k-i}\\
&= c_k.
\end{aligned}$$

Using these equalities we can simplify the code by computing the left hand side of equation (4.6), as shown by Figure 4.7. Note that there is no need to implement the final subtraction between the two summations. The signs of the coefficients of $\tilde{b}$ are determined by the indices. The subtraction is avoided by accessing the coefficients of index greater than or equal to n. Moreover, the loop bound is constant and no useless multiplications are computed (as was the case for `schoolbook1`). Multiple simultaneous memory accesses have also disappeared from the code, so that the inner loop can be fully pipelined and parallelized if requested. The resulting implementation should therefore be able to compute the n^2 multiplications in about n^2 clock cycles, with high frequency.

A quick experiment with the HLS tool shows that for `schoolbook1b` we can obtain a parallelized implementation with 6 DSP blocks and less than 500 LUTs executing in less than 40,000 cycles with a clock period of 1.87 ns. This performance is equivalent to the fastest result in Table 4.2, but the area usage is improved: 4 less DSP blocks are needed. The simplification of the source code allows to better pipeline the inner loop, resulting in a higher obtained frequency for smaller architectures. The area usage can be reduced even further by having a look at the reduction algorithm. Indeed, the Barrett algorithm relies heavily on integer multiplication, and the increase in the number of DSP blocks is linear in the degree of parallelization. We replace the Barrett reduction by the modular reduction algorithm for moduli of the form $2^k - 2^l + 1$ using only shift and add operations, described in the state of the art.

Another way to reduce the number of DSP blocks in the implementation is to reuse the DSP block for both multiplications in the Barrett algorithm. This can be accomplished using the **allocation** directive. The best HLS results using this directive are shown in Table 4.3.

The decryption in RLWE schemes consists of a polynomial multiplication followed by a subtraction. By far the most of the ressources and computation time are consumed by the polynomial multiplication, so that it makes sense to compare our results with implementations of the decryption algorithm. A few hundred cycles and some LUTs should be added to our architectures to make the comparison credible.

In most cases, like in [105], the decryption function is implemented using the NTT, but [97] is an exception. Their goal is to optimize the area utilization of the implementation, and to this end they prefer to schoolbook algorithm over the NTT. They use a Spartan 6 FPGA and choose parameter set $(q, n) = (4096, 256)$. We implement our multiplication algorithm for this parameter set as well. The advantage of picking $q = 4096$, is that this is a power of 2, so that modular reduction does not need to be computed. The results are shown in Table 4.3. The implementations found in the literature do not seem to take much advantage of parallelism: often only one multiplication unit is used.

It seems better to use constant loop bounds instead of variable loop bounds, although this means that more information (longer vectors) has to be stored in the memory. For application in cryptography this should not make much of a difference: the polynomial that is stored on the device (the secret key) has very small coefficients. An improvement is given by the optimized version presented in this section.

The shift and add based reduction algorithm seems to be better (smaller/faster) than the Barrett

Table 4.3: Comparing our results from HLS to those reported in literature.

source	algorithm	parameters q, n	FPGA	latency (cycles)	freq. MHz	time (μs)	DSP	LUT
this work	schoolbook	4096, 256	Artix-7	67585	283	239	1	294
this work	schoolbook	4096, 256	Artix-7	35329	361	98	2	279
[97]	decrypt (schoolbook)	4096, 256	Spartan-6	66338	189	351	1	112
this work	schoolbook (shift add)	7681, 256	Artix-7	68609	283	242	1	322
this work	schoolbook (shift add)	7681, 256	Artix-7	37377	508	74	2	412
this work	schoolbook (Barrett)	7681, 256	Artix-7	67585	191	354	1	405
this work	schoolbook (Barrett)	7681, 256	Artix-7	72706	534	136	2	499
this work	schoolbook (Barrett)	7681, 256	Artix-7	37634	508	74	4	346
[105]	decrypt (NTT)	7681, 256	Virtex-6	2800	313	9	1	1349

reducer for our implementation of the schoolbook algorithm, as can be seen in the table above. Although the DSP block in the Barrett reducer can be shared with the coefficient multiplier, this works only on relatively low frequencies. For higher clock frequencies, more DSPs are needed. Interestingly, the performance of our schoolbook implementation using the shift and add based reduction algorithm is similar to that of the implementation using the parameter set where $q = 4096$. While a few more LUTs are needed, the implementation of the shift and add based reduction algorithm does not have much impact on the latency, compared to the implementation with power of 2 modular reduction. The pipeline is not slowed down by the shift and add based reduction.

We use twice as many LUTs as [97], but with a similar latency and higher frequency, our implementation is faster. By applying the **unroll** directive, we get an even faster implementation, making use of parallelism. However, the implementations using the NTT are still a lot (10 to 20 times) faster than the schoolbook multiplier. For efficient RLWE implementations, the NTT algorithm is indispensable.

Chapter 5

LWE, RLWE and MLWE on FPGA

5.1 Introduction

In this chapter, we discuss the implementation of LWE, RLWE and MLWE based public key encryption on FPGA and show how to speed up the computations using parallelism. We implement CPA secure algorithms 8 and 9 from the state of the art. More precisely, we use High Level Synthesis to:

- implement LWE, RLWE and MLWE-based PKEs for parameter sets used by NIST round 2 candidates Frodo [28], NewHope [4] and the original Kyber [27] scheme;
- study the speed-up that can be obtained by parallelizing the computation of the critical operations;
- implement the Fujisaki-Okamoto transform to obtain CCA secure key exchange mechanisms (KEM);
- analyse the impact of the PRNG choice and the FPGA target on the performance and cost of MLWE-based PKE.

Using the same techniques, the same effort and the same FPGA for each implementation, we provide an objective comparison between the performances of the different algorithms. The implementations obtained by using HLS have, in many cases, a better area/computation time trade-off than the results found in other works using VHDL/Verilog. Our MLWE implementations use up to 28 times fewer DSP blocks than the results from [44], while computing the encapsulation algorithm 4% faster. At the same time, the effort needed to explore a range of architectures is much reduced, compared to VHDL/Verilog techniques. We study the impact of the PRNG on the computation time of the encryption algorithm. We consider a fast implementation using the lightweight Trivium PRNG [46], a slower but more secure solution using the SHA3 based SHAKE-256 [23] and a hybrid one using both PRNGs. This hybrid solution uses Trivium to generate the pseudorandom part of the public key, while using

SHAKE-256 to sample the secret error terms during encryption. We discuss and implement rejection sampling for the pseudorandom part of the public key. We implement the Fujisaki-Okamoto transform [56] to obtain CCA secure architectures for LWE, RLWE and MLWE. To the best of our knowledge, this is the first CCA secure FPGA implementation of MLWE based encryption. The implementation by [44] implements MLWR, which does not use the NTT for polynomial arithmetic.

State of the art implementations and their algorithms are discused in section 5.2. In sections 5.3, 5.4 and 5.5 respectively, the details of our LWE, RLWE and MLWE implementations are explained, with a focus on parallel computation. Implementations of the CPA to CCA transformation (algorithms 6 and 7), and different methods of randomness generation are discussed in section 5.6. A comparison of our results with other works is provided in section 5.7.

5.2 Implementation of main operations

Matrix Multiplication The multiplication of the public key $\mathbf{A}$ with the error matrix $\mathbf{e}_1$ is the most expensive operation in the standard LWE scheme. It consists of k^2m multiplications in the rings $\mathbb{Z}_{2^{15}}$ or $\mathbb{Z}_{2^{16}}$. In [64] the matrix multiplication is accelerated by computing partial products in parallel using up to 16 DSP blocks.

Polynomial Multiplication In RLWE and MLWE, the most expensive arithmetic operation is the polynomial multiplication. Multiplication in the ring $\mathcal{R}_q$ is computed using the Number Theoretic Transform (NTT). FPGA implementations of NewHope using the NTT for $n = 1024$ are given by [90] and [74]. A fast and area optimized implementation for $n = 256$ is given by [105]. While implementations using Schoolbook polynomial multiplication have been proposed [97][77], they are much slower than the NTT. To compute a polynomial multiplication using the NTT, the polynomials should be mapped to the NTT domain where the polynomial multiplication is a point-wise operation taking only n modular multiplications in $\mathbb{Z}_q$. Addition and subtraction can also be performed point-wise in the NTT domain. The inverse NTT is applied to bring the result back in the time domain.

The transform is efficiently computed in $\log_2(n)$ stages of $\frac{n}{2}$ multiplications using the Cooley-Tukey algorithm [40]. Using the constant geometry variant [92] of the NTT algorithm, the memory access pattern is independent of the stage. The values are not read from the same memory as the one that the updated variables are written to, therefore 2 BRAMs are needed in the implementation. At each iteration of the stage loop, 2 values are read from the memory, a butterfly operation is computed and the 2 results are written to the memory. A detailed description of the stage is given by Algorithm 13. All arithmetic operations are performed in $\mathbb{Z}_q$.

In [77] the Schoolbook algorithm is implemented, using an optimized algorithm for coefficient multiplication. A Xilinx DSP48E block can be used for 18 × 25-bit integer multiplication. The coefficients sampled from the error distribution can be written on a few bits. Therefore, a naive

Algorithm 13 i-th stage of the NTT [92]

```
function STAGE($X, i$)
    for $j \leftarrow 0$ to $\frac{n}{2} - 1$ do
        $\theta \leftarrow \omega^{\lfloor \frac{j}{2^i} \rfloor \cdot 2^i}$    ▷ Get twiddle factor from memory
        $(x_0, x_1) \leftarrow (X[2j], X[2j+1])$    ▷ Read from memory $X$
        $\big(Y[j], Y[j+\frac{n}{2}]\big) \leftarrow (x_0 + x_1, (x_0 - x_1)\theta)$    ▷ Write to memory $Y$
    return $Y$
```

multiplication of a coefficient of $w = \lfloor \log q \rfloor + 1$ bits with an error coefficient would not make optimal use of the DSP block. The implementation from [77] takes two error coefficients e_0, e_1 of size $w_\lambda = 1 + \lfloor \log_2 \lambda \rfloor$ and defines a new $(w + 2w_\lambda)$-bit coefficient $e_0 + 2^{w+w_\lambda} e_1$. If $(w + 2w_\lambda) < 25$, then for any w-bit coefficient a the multiplication $(e_0 + 2^{w+w_\lambda} e_1)a$ can be computed on one DSP block. The first product $e_0 a$ can be read on the first $w + w_\lambda$ bits of the output, starting with the LSB. The second product $e_1 a$ is obtained by applying $w + w_\lambda$ left shifts to the output and again selecting the first $w + w_\lambda$ bits of the remainder. The sign of the products is computed separately. Then two multiplications are obtained for the price of one.

Binomial sampling The $\mathcal{B}_\lambda(\mathbb{Z}_q)$ distribution is sampled by generating 2λ random bits $x_1, \ldots, x_\lambda$ and $y_1, \ldots, y_\lambda$ and computing $\sum_{i=1}^{2\lambda} x_i - y_i \bmod q$. The sampling requires 2λ random bits per coefficient. For a total of $mn(2k+m)$ coefficients for the 3 errors $\mathbf{e}_1, \mathbf{e}_2$ and $\mathbf{e}_3$, the amount of random bits needed is considerable. In the specifications of most of the NIST round 2 candidates it is suggested to use SHAKE-256 or AES to supply the randomness. Some implementations however, such as [64], use Trivium because it is faster. Precomputing random bits and storing them in BRAM is used in [65] to improve the throughput of the PRNG.

Modular reduction for $q = 2^w - 2^l + 1$ In LWE based cryptography, the modulus q can be of some specific form such that for $x < q^2$ it is easy to compute $x \bmod q$. In the rings $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_{2^m+1}$ for some integer m for example, modular reduction can be computed using bit-wise operations and one addition. Any $2n$-bit integer a can be written as $a = a_0 + 2^n a_1$ for some $a_0, a_1 < 2^n$. Then $a \bmod 2^n = a_0 + 2^n a_1 \equiv a_0 \bmod 2^n$. The modular reduction for $q = 2^n$ can be computed directly by taking the least significant bits of a. Similarly, for $q = 2^n + 1$, the equation $a_0 + 2^n a_1 \equiv a_0 - a_1 \bmod (2^n + 1)$ leads to a direct and simple algorithm for modular reduction.

In order to use the NTT however, one needs q to be a prime for which $q \equiv 1 \bmod 2n$ such that there exists a $2n$-th root of unity in $\mathbb{Z}_q$. The choice is therefore limited. For prime moduli of the form $q = 2^w - 2^l + 1$ for some integers w and l and for $l \geq \log_2(2n)$, it holds that $2^w - 2^l + 1 \equiv 1 \bmod 2n$. For $n = 256$ suitable primes include $7681 = 2^{13} - 2^9 + 1$ which is used in Kyber [27]. For moduli of the form $2^w - 2^l + 1$, fast reduction algorithms in the style of [109] can be used. We use algorithm 4.2 described in chapter 4.

5.3 FPGA Implementation of LWE

5.3.1 Parameters used in the implementations

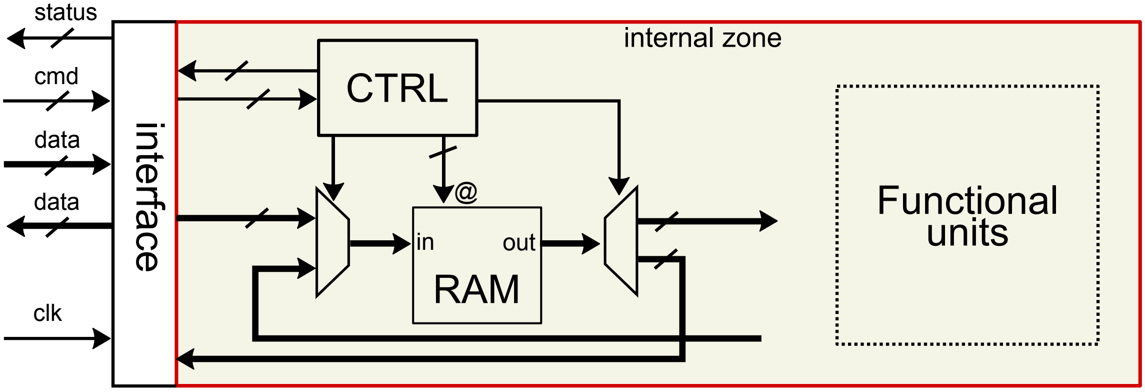
We implement the CPA and CCA secure LWE, RLWE and MLWE schemes for parameter sets shown in Table 5.1. We choose LWE parameters from FrodoKEM [28] except for the Gaussian distribution. We sample the $\mathcal{B}_\lambda$ distribution instead, where λ is chosen such that the obtained $\mathcal{B}_\lambda$ distributions are close to the Gaussian distributions from FrodoKEM. This allows us to make a fair comparison between LWE on one hand and RLWE and MLWE (both using binomial distributions) on the other. To the best of our knowledge, there does not exist any attack that exploits the small difference between the sampled distribution and the Gaussian distribution used in the security proof. The performance of the best algorithms solving LWE does not depend on the exact error distribution, which is why schemes such as Kyber [27] also prefer binomial sampling. The parameters for RLWE correspond to those used by NewHope. A newer version of Kyber proposes to use the modulus $q = 3329$. On FPGA, there is hardly any gain in computation time to be expected from replacing 13-bit operands by 12-bit, because in both cases an integer multiplication can easily be computed on a single DSP48E. Reducing the operand size however, comes at the cost of having to implement quadratic extension field arithmetic. To avoid the overhead in computation time that this would cause, we choose to implement the original scheme using $q = 7681$. The different parameter sets are designed for the security levels 1, 3 and 5 defined by the NIST, where 1 correspond to AES-128, 3 to AES-192 and 5 to AES-256. Security level 1 is claimed by [28] for Frodo using parameter set LWE-640, and by [27] for Kyber using parameters set MLWE-512. Security category 3 proposals use parameter sets LWE-976 and MLWE-768, while LWE-1344, RLWE-1024 and MLWE-1024 are used in security category 5.

Table 5.1: Parameter sets used in our implementations.

Algorithm	n	m	k	q	λ
LWE	1	8	640/976/1344	$2^{15}/2^{16}/2^{16}$	15/10/4
RLWE	256/1024	1	1	7681/ 12289	3/8
MLWE	256	1	2/3/4	7681	5/4/3

Figure 5.1 presents the high-level architecture of our *accelerator*. For encryption, the public keys are first loaded into the local *RAM*, then the computations are performed by the *functional units*. During encryption/decryption our accelerator is isolated for security reasons, it does not take any input or generate any output. After encryption/decryption, the result is sent out through the interface. In this work, all the communications through the interface are included in our results. Depending on parameter n, the typical time spent for interfacing represents about 12% to 21% of the total encryption/decryption time.

Figure 5.1: High level architecture of our accelerator.



5.3.2 Matrix arithmetic for LWE

We extend the method in [77] to speed-up schoolbook polynomial multiplication, described in Sec. 5.2, to the matrix multiplication for the standard LWE scheme. Matrices $\mathbf{A}$ and $\mathbf{e}_1$ coefficients are 15 and $w_\lambda = 1 + \lfloor \log_2 \lambda \rfloor$ bits wide respectively. We pack two coefficients $e_{00}||e_{10}$ to reduce the $8 \times k$ matrix $\mathbf{e}_1$ with w_λ-bit elements to a $4 \times k$ matrix with $(w + 2w_\lambda)$-bit ones. Then multiplying one coefficient from $\mathbf{A}$ by one from $\mathbf{e}_1$ requires a single DSP block.

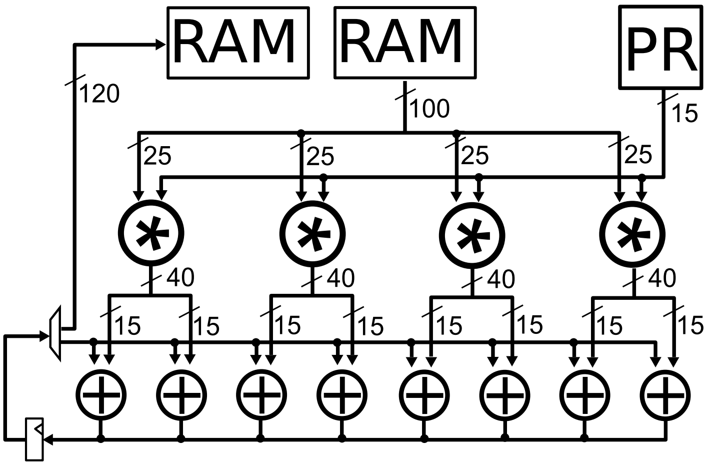
The coefficients of the public key matrix $\mathbf{A}$ are generated using the PRNG. At each clock cycle, one coefficient is generated. During the first clock cycle, a_{00} is generated and multiplied by all 4 coefficients in the first column vector of $\mathbf{e}_1$.

$$\underbrace{\begin{pmatrix} e_{00}||e_{10} & e_{01}||e_{11} & \cdots \\ \vdots & \vdots & \\ e_{60}||e_{70} & e_{61}||e_{71} & \end{pmatrix}}_{k \text{ columns}} \times \underbrace{\begin{pmatrix} a_{00} & a_{01} & \cdots \\ a_{10} & a_{11} & \\ \vdots & & \\ \vdots & & \\ a_{(k-1)0} & & \end{pmatrix}}_{k \text{ columns}}$$

$$= \underbrace{\begin{pmatrix} c_{00} & c_{01} & \cdots \\ c_{10} & & \\ \vdots & \vdots & \\ c_{70} & & \end{pmatrix}}_{k \text{ columns}} \tag{5.1}$$

The resulting vector is added to the first column vector of the output matrix. All the coefficients that are loaded in the first clock cycle are coloured blue in equation (5.1). During the second clock cycle, the red coefficients are loaded. The resulting integer products are all added to the first column

Figure 5.2: Architecture for matrix multiplication $\mathbf{e}_1^\intercal\mathbf{A}$. The 4 DSPs compute 8 integer products. Coefficients of $\mathbf{A}$ are generated by the PRNG ("PR").



vector of the output matrix. The first column vector of this output matrix is completely computed after k (+ pipeline depth) clock cycles. Only then the computation of the second column vector begins.

The row vectors of $\mathbf{e}_1$ are implemented on 1 BRAM each, so that the matrix $\mathbf{e}_1$ uses 4 BRAMs. The architecture of the matrix multiplication is shown in figure 5.2. To increase the level of parallelism by a factor two, the blue and red multiplications can be performed at the same time. Then twice as many DSP blocks are required for the matrix multiplication and two coefficients of $\mathbf{A}$ have to be generated at the same time. For higher degrees of parallelism, multiple elements on the same row vectors of $\mathbf{e}_1$ have to be read simultaneously. Therefore the row vectors of $\mathbf{e}_1$ have to be implemented on multiple BRAMs each.

5.3.3 Parallelization using HLS

We use Xilinx Vivado HLS (version 2018.1) to generate architectures for the target FPGA XC7A200. The C source code of the matrix multiplication $\mathbf{c}_1 \leftarrow \mathbf{e}_1^\intercal\mathbf{A}$, described in equation 5.1, is shown in Figure 5.3. The loops labelled `col_A` and `row_A` iterate over the columns and rows of A respectively. Column vectors of the output matrix are loaded and stored by loops `copy1` and `copy2`. The `prng` function generates the next coefficient of A, and `compute_2products` computes $a \cdot (e||e')$ for coefficients a, e, e' using the error encoding method described in the previous paragraph.

In order to specify how this code should be implemented by the HLS tool, the *directives* can be applied to parts of the code. Applying the `pipeline` directive to the loop `row_A`, ensures that this loop is pipelined and the subloop `row_E` is completely unrolled. That is, all 4 iterations of the loop `row_E` are computed at the same time on 4 DSPs. Arrays are implemented on a single BRAM by default. Without any specifications, the HLS tool would try to implement `E1` on a single BRAM. However, all 4 elements of each column vector have to be loaded simultaneously. Therefore we implement `E1` on

Figure 5.3: Source code for matrix multiplication $C_1 \leftarrow E_1^\intercal A$.

```
col_A: for(i=0; i<k; i++){
    copy1: for(ii=0; ii<8; ii++)
        C1_tmp[ii] = C1[ii][i];                // copy from BRAM to registers
    row_A: for(jj=0; jj<k; jj++){
        sum = 0;
        prng(State_A, &a_coeff);                   // PK coefficient from PRNG
        row_E: for(j=0; j<4; j++){
            compute_2products(a_coeff, E1[j][jj], &prod1, &prod2);
            C1_tmp[2*j] = C1_tmp[2*j] + prod1;
            C1_tmp[2*j+1] = C1_tmp[2*j+1] + prod2;      // update output matrix
        }
    }
    copy2 :for(ii=0; ii<8; ii++)
        C1[ii][i] = C1_tmp[ii];                    // copy from registers to BRAM
}
```

4 different BRAMs by setting the directive `array_partition`. This directive partitions an array into multiple smaller arrays, which are then implemented on multiple BRAMs.

We parallelize the computation even further by applying the directive `unroll` on loop `row_A`. When using this directive, the unroll factor has to be specified. Our implementation results are obtained by setting the unroll factor to 2, 4, 8 and 16. For unroll factors 4, 8 and 16, multiple elements on the same row have to be accessed at the same time. Therefore the array `E1` has to be partitioned in the second dimension as well, using the `array_partition` directive, to prevent simultaneous accesses to the same BRAM.

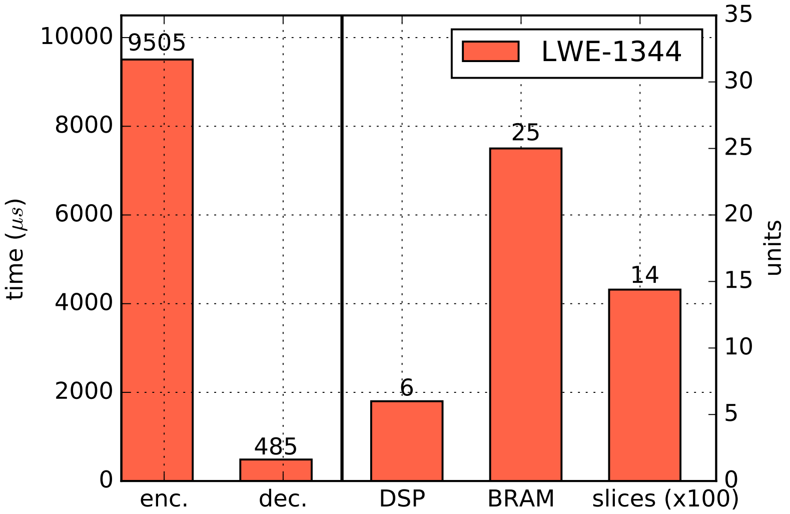
Table 5.2: LWE implementations results for encryption and decryption.

Algorithm	Freq.	Time (enc/dec)	Area
LWE- k	MHz	μs	DSP, BRAM, Slices, LUT
LWE-640	200	2275/232	6, 16, 1629, 4311
LWE-976	200	5123/353	8, 16, 1601, 4322
LWE-1344	200	9506/486	6, 25, 1439, 3832

5.3.4 Implementation results

The implementation results are obtained using Vivado HLS (version 2018.1) for target device Artix xc7a200, and are shown in table 5.2. The error encoding technique packing two error terms in one $w+2w_\lambda$ bit integer allows to compute 8 multiplications in parallel using 4 DSP blocks for the parameters

Figure 5.4: Computation time and area utilization for our LWE-1344 implementation.



sets of $k = 640$ and $k = 1344$. For $k = 976$ however, the error terms are still 5 bit integers while the coefficient size is increased to 16 bits (see parameter sets in Table 5.1). Therefore $w + w_\lambda > 25$ and extra DSP blocks are needed for the multiplications. For $k = 1344$ the size of the error terms decreases to 4 bits.

The matrix multiplication using 4 parallel running DSP blocks is computed in roughly $k^2 = 409600$ cycles for $k = 640$. This operation takes up 90 percent of the total encryption time. A visualization of the impact of parallelism on the timing and area results of the implementation is shown in Figure 5.5. These results are for the encryption algorithm only. It can be seen that for unroll factor 2× the total encryption time is divided by almost 2. The relative cost increase in terms of DSPs is lower than the factor by which the computation time decreases. In terms of slices, LUTs and BRAMs, the trade-off is even more favourable for the parallelized implementation, which even holds for the more parallelized implementations using unroll factors 4×, 8× and 16×. For these higher degrees of parallelism however, the number of DSPs increases faster than the computation time decreases. The 8× and 16× implementations run on smaller frequencies, limiting the obtained speed-up. More detailed results including comparisons with results from the state of the art are given in Table 5.7 in section 5.7

5.4 RLWE Implementations

In this section, we present our implementation of the RLWE based encryption and decryption algorithms described in the state of the art. One of our goals is to show that competitive results can be obtained using HLS from C code for a reduced design cost compared to VHDL or Verilog design.

Figure 5.5: Comparing the base implementation (blue) to parallelized versions using unroll factors 2 (green), 4 (red), 8 (cyan) and 16 (magenta).

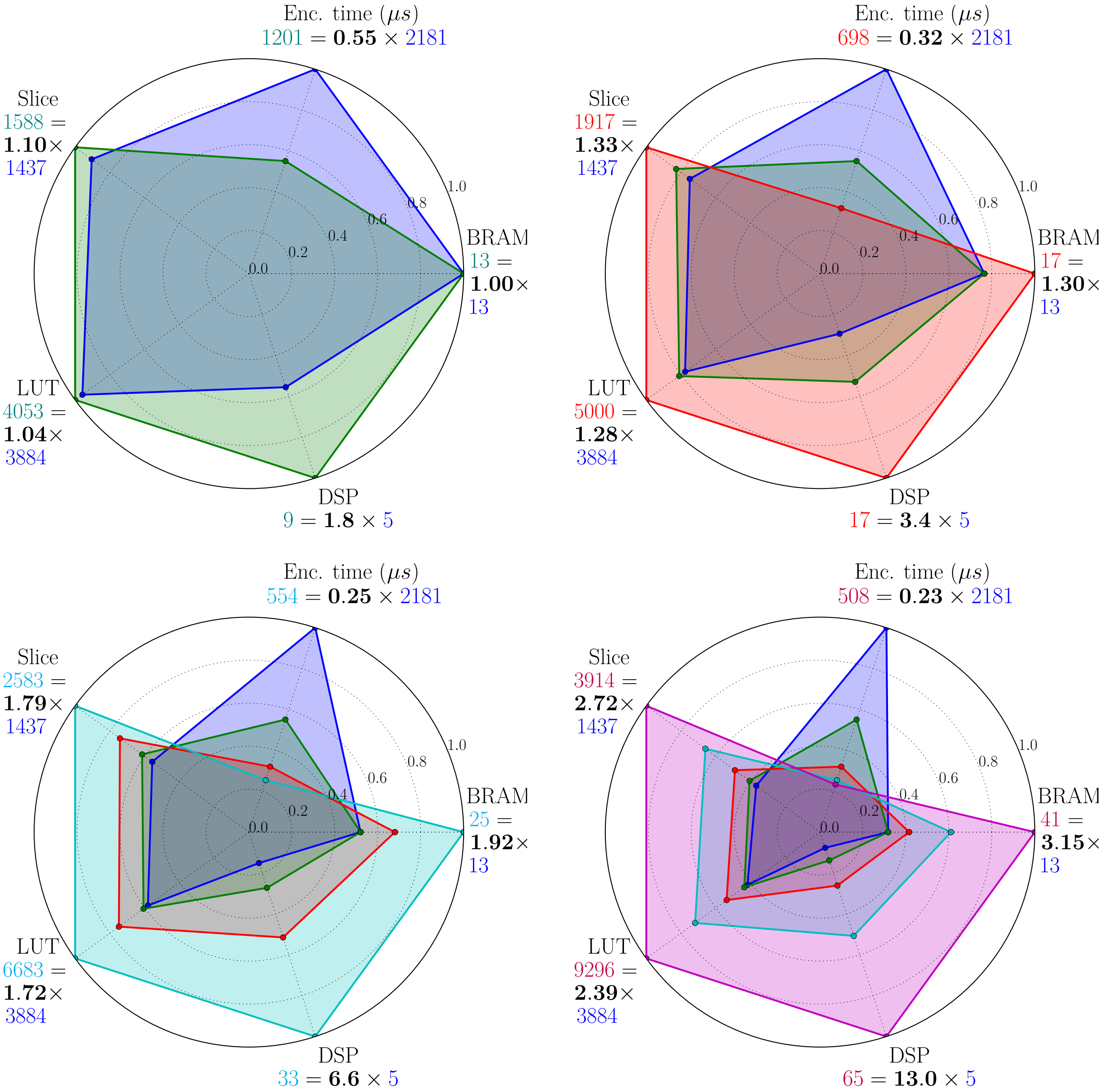
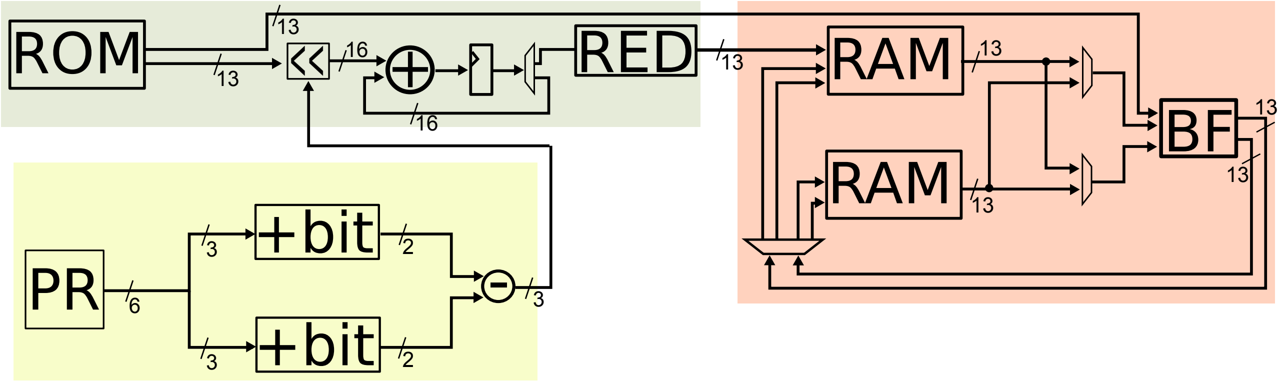


Figure 5.6: Architecture computing the error polynomials in the NTT domain. The yellow part uses the PRNG (TR unit) to generate samples from the $\mathcal{B}_\lambda$ distribution (here: $\lambda = 3$). The +bit operator computes the sum of λ input bits. The NWC (upper left) is computed using a shift-based multiplier and modular reduction (RED). The NTT is computed on the right, using one Gentlemen-Sande butterfly (BF) operator.



Parameters. In order to compare with literature results, we implement RLWE for the parameter sets $(n, q, \lambda) = (1024, 12289, 8)$ and $(256, 7681, 3)$. For $n = 1024$, the implemented algorithm is a simplification of the CPA-secure version of NewHope1024 PKE with key reuse. We do not implement the key refreshing, ciphertext compression/decompression and key encoding/decoding. For simplicity we use the Trivium stream cipher as PRNG.

Encryption and Decryption. Following [98], we avoid the bit-reversal step by implementing both the DIF and DIT NTT. The stage loop is fully pipelined, such that it takes just over $\frac{n}{2}$ clock cycles to complete one stage. The complete forward transformation is computed in few more than $\frac{n}{2} \log n$ cycles.

The error polynomials $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ are sampled from the binomial distribution $\mathcal{B}_\lambda(\mathcal{R}_q)$. The required random bits are provided by the PRNG. Since the ciphertext part $\mathbf{c_1} = \mathbf{a}\mathbf{e}_1 + \mathbf{e}_2$ will be sent in the NTT domain, the NTT has to be applied to both $\mathbf{e}_1$ and $\mathbf{e}_2$. The NWC must be computed for both polynomials. To compute these multiplications, we use the fact that the coefficients are sampled from $\mathcal{B}_\lambda$ and therefore are bounded by $-\lambda$ and λ. The multiplications can be computed using only a few shifts and additions, without using a DSP block. The NTT is then applied to $\mathbf{e}_1$ and $\mathbf{e}_2$ simultaneously, using two parallel NTT units each consisting of one butterfly unit and two BRAMs. The architecture for sampling $\mathbf{e}_1$ (or $\mathbf{e}_2$) and mapping it to the NTT domain is shown in Figure 5.6.

The architecture for decryption is shown in Figure 5.7. The area and timing implementation results for RLWE are shown in Table 5.3 with similar solutions from the literature.

Our small implementation is denoted by V1. This implementation with only 1 DSP block is comparable in size and speed to [96] but is larger and 15 to 20% slower than the cryptoprocessor from

Table 5.3: FPGA implementation results for our RLWE solutions (denoted V1, V2 and V3) and literature solutions. If specified, Encryption/Decryption and Server/Client/Server (for a 3-step key exchange) timing results are shown separately. Separate area results for Server and Client are indicated with +.

Source	FPGA	Latency (clock cycles)	MHz	Time (μs)	Slice, DSP, LUT, BRAM
$n = 256$					
[96]	XC6VLX75	6861/4404	262	26.2/16.8	1506, 1, 4549, 12
[105]	XC6VLX75	6300/2800	313	20.1/9.1	n.a., 1, 1349, 2
V1	XC7A200	5039/2188	208	24.2/10.5	1624, 1, 4365, 5
V2	XC7A200	3764/2239	250	15.1/9.0	2122, 6, 5616, 8
$n = 1024$					
[74]	XC7Z020	6900/10300/2800	133/131	51.9/78.6/21.1	n.a., (8+8), (18756+20826), (14+14)
[90]	XC7A035	171124/179292	125/117	1369/1532	0, (2+2), (5142+4498), (4+4)
V3	XC7A200	16146/9586	250	64.6/38.3	4106, 7, 11164, 12

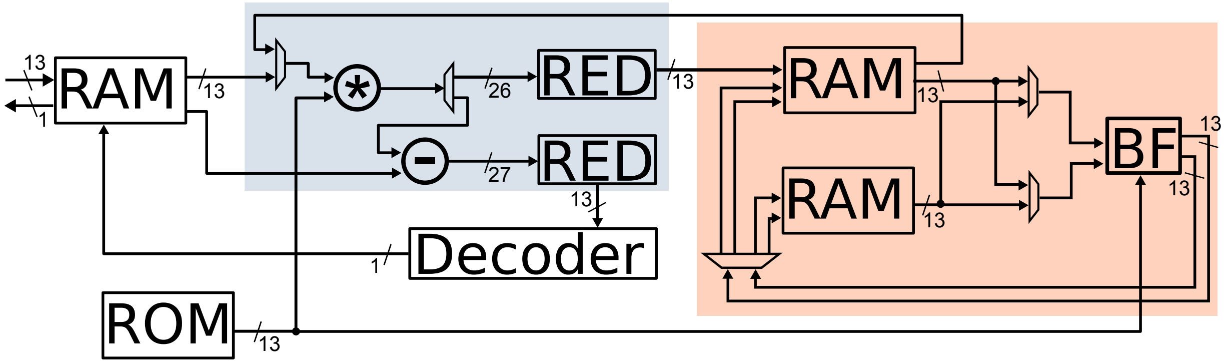
[105]. By computing the forward NTTs in parallel in V2, we are faster than both, but more DSP blocks are needed. For $n = 1024$, the key exchange implementation by [74] is comparable with our V3 results in terms of speed, but the V3 implementation uses 50% less DSP blocks and BRAMs. We conclude that results obtained using HLS are comparable or, in the best case, even better in terms of speed (up to 25%) and/or area (up to 50%) than results from works based on VHDL or Verilog implementations.

5.4.1 Optimizing the area utilization

From here on, we will focus on the RLWE implementation for $n = 1024$ only, since this parameter set is used by NewHope (for the highest security category). The parameter set for $n = 256$ has been used in several implementations in the state of the art, but is considered not secure enough for RLWE based cryptosystems and none of the PKE submissions in the NIST standardization competition uses this parameter set.

The NTT is used in both encryption and decryption. During encryption, the forward NTT is computed while during decryption the inverse transformation is computed. By sharing the ressources used for these two operations, area utilization can be decreased. Encryption and decryption can then be computed in the same time as in table 5.3, but using only 4 DSPs instead of 7 (for $n = 1024$).

Figure 5.7: Architecture for the decryption. The ciphertext is completely loaded in the RAM before starting the computations. The two pointwise operations (one before and one after the inverse NTT) in the blue region share a DSP block.



Using this optimization, we obtain implementation results for 3 different degrees of parallelism (table 5.4). The first architecture uses only one DSP block and no parallelization is used. By computing the two forward NTTs simultaneously, $\frac{n}{2}\log n$ less cycles are needed for the encryption function, resulting in a smaller latency. The second architecture also runs on a higher frequency than the first, resulting in an even more significant speed-up. The third architecture is obtained by unrolling loops of the point-wise computations by a factor 2. The gain in speed, compared to the second architecture, does not seem worth the cost of the extra DSPs and BRAMs.

Table 5.4: RLWE-1024 encryption/decryption results.

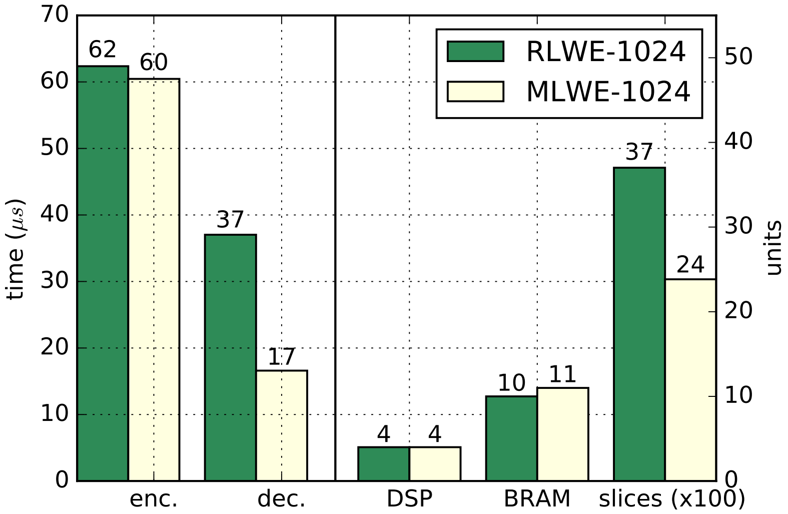
	Freq.	Time (enc/dec)	Area
Version	MHz	μs	DSP, BRAM, Slices, LUT
Sequential	206	110/47	1, 11, 3820, 10563
NTTs in parallel	258	63/38	4, 10, 3701, 10112
Unroll	251	59/35	6, 16, 4474, 12301

5.5 MLWE implementations and comparison

5.5.1 Modifying the RLWE implementation

We transform our RLWE-1024 implementation for MLWE using slight changes, starting by changing n from 1024 to 256. The arithmetic units are re-used for computations in $\mathcal{R}_q$ and MLWE. This includes our architecture in Figure 5.6 (modified for $n = 256$) that generates binomial samples in the NTT domain and now denoted BN. The same operations are performed but on polynomial coefficients of k-dimensional vectors over $\mathcal{R}_q$. The MLWE scheme is thus implemented by applying the operators used in RLWE to each of the k polynomials (each of degree n) of the vectors in a sequential manner.

Figure 5.8: Comparison of MLWE with RLWE FPGA implementations for the same security level.



This is achieved by modifying the control accordingly. Each vector consists of $14 \cdot 256 \cdot k$ bits and is stored in one 18 kb BRAM. For $k = 4$, around 14 kb is used in each BRAM, while for $k = 2$, only 7 kb is used. In a sequential architecture, the number of BRAMs is the same for $k \in \{2, 3, 4\}$. Extra additions and a modified control are needed to support the multiplication of matrices and vectors of dimension k. To avoid storing the $k \times k$ random matrix $\mathbf{A}$, which is part of the public key, we use the PRNG to generate the coefficients of the polynomials in matrix $\mathbf{A}$ on the fly, as suggested by [27]. The public key to be stored in the architecture only consists of the vector $\mathbf{b} \in \mathcal{R}_q^k$ and the seed for the PRNG. We apply one step of rejection sampling in order to avoid too much bias in the distribution of the coefficients (see section 5.6), as proposed for instance for Kyber in [27].

The parameter k is used as a security parameter and determines the number of multiplications in $\mathcal{R}_q$ to be computed. During the encryption, $k^2 + k$ multiplications in $\mathcal{R}_q$ and $2k$ forward NTTs are needed. The decryption consists of k multiplications in $\mathcal{R}_q$, while there is only one inverse NTT, independently of k.

We compare the performance of the RLWE scheme for $n = 1024$ with the performance of MLWE for $k = 4$. It can be seen in table 5.5 and figure 5.8 that the encryption time does not differ by much although RLWE seems to be slightly slower. More significantly, the MLWE scheme decrypts twice as fast as RLWE. The impact of the parameter k on the decryption time of MLWE is limited, since only the size of the computation $\mathbf{c}_1 \cdot \mathbf{s}$ depends on the parameter k. During the encryption however $k^2 + k$ multiplications in $\mathcal{R}_q$ and $2k$ NTTs have to be computed. The encryption time is therefore heavily impacted by increasing the parameter k.

Table 5.5: MLWE FPGA implementations for different security levels.

Algorithm MLWE- $(k \times n)$	Freq. MHz	Time (enc/dec) μs	Area DSP, BRAM, Slices, LUT
MLWE-512	256	30/12	4, 11, 2380, 5538
MLWE-768	256	44/15	4, 11, 2540, 6031
MLWE-1024	250	61/17	4, 11, 2383, 5515

5.5.2 Parallelization of operations in $\mathcal{R}_q^k$

We also propose parallelized implementations that compute MLWE encryption and decryption for different values of the security parameter k (vector length) in almost the same time. While the computations in $\mathbb{Z}_q$ are still performed sequentially, hardware is added to compute the operations on a higher level (matrix-vector operations) in parallel. During the encryption the k components of the error vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ have to be sent to the NTT domain. All of these $2k$ transforms are computed simultaneously. The operation (for $k = 2$)

$$\mathbf{e}_1, \mathbf{e}_2 \xmapsto[\frac{n}{2}\log(n) \text{ cycles}]{} \begin{pmatrix} \mathrm{NTT}(\mathbf{e}_1^{(0)}) \\ \mathrm{NTT}(\mathbf{e}_1^{(1)}) \end{pmatrix}, \begin{pmatrix} \mathrm{NTT}(\mathbf{e}_2^{(0)}) \\ \mathrm{NTT}(\mathbf{e}_2^{(1)}) \end{pmatrix}$$

is computed in the time it takes to compute one NTT, that is, $\frac{n}{2}\log(n) + \delta$ cycles for where δ is the pipeline depth. Similarly, PRNGs and binomial samplers are added to sample the $2k$ error polynomials simultaneously. The k^2 multiplications in $\mathcal{R}_q$ for the computation of $\mathbf{c}_1 \leftarrow \mathbf{A}^\intercal \mathbf{e}_1 + \mathbf{e}_2$ and the k multiplications in $\mathcal{R}_q$ needed to compute $\mathbf{c}_2$ are also computed in parallel. For $k = 2$, the operation

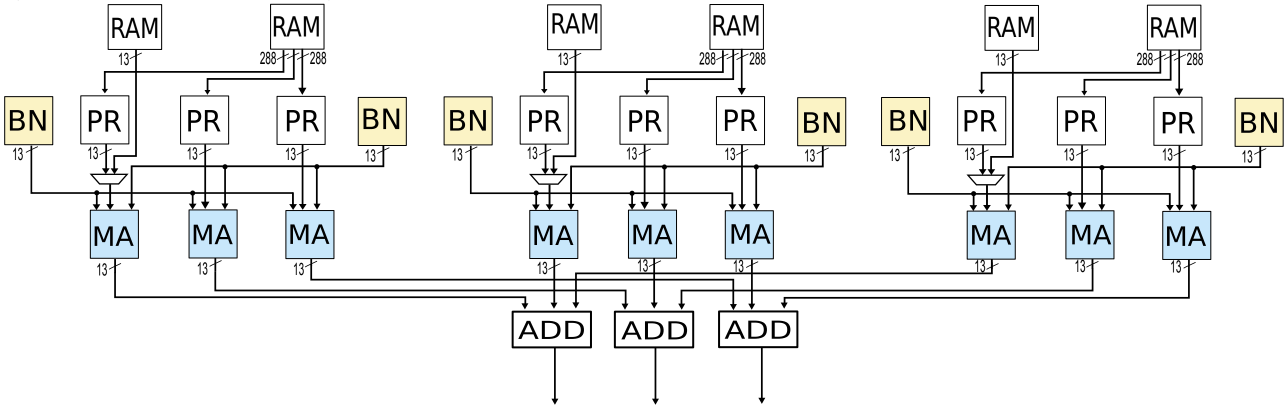
$$\mathbf{e}_1, \mathbf{A} \xmapsto[n \text{ cycles}]{} \begin{pmatrix} \mathbf{a}^{(00)} \odot \mathbf{e}_1^{(0)} + \mathbf{a}^{(01)} \odot \mathbf{e}_1^{(1)} \\ \mathbf{a}^{(10)} \odot \mathbf{e}_1^{(0)} + \mathbf{a}^{(11)} \odot \mathbf{e}_1^{(1)} \end{pmatrix}$$

is computed in just over n cycles, which is the time it takes to compute one single point-wise multiplication. For the computation of $\mathbf{A}^\intercal \mathbf{e}_1$, in order to compute the k^2 multiplications over $\mathcal{R}_q$ in parallel, we need to be able to access all k^2 coefficients of $\mathbf{A}$ at the same time. Therefore, we generate a seed for the PRNG for each of the k^2 coefficients of $\mathbf{A}$. The public key then consists of a vector $\mathbf{b} \in \mathcal{R}_q^k$ and a seed for the PRNG that is used to generate the k^2 seeds for the $k \times k$ matrix $\mathbf{A}$. The parallel architecture for $k = 3$ using the BN unit described in the previous section is shown in figure 5.9.

5.5.3 Parallelization using HLS

The C source code in figure 5.10 is an excerpt from the MLWE encryption implementation. This code computes the matrix-vector product $\mathbf{A}^\intercal \mathbf{e}_1$ where all the matrix and vector coefficients are in the NTT

Figure 5.9: Proposed parallel architecture for the matrix-vector multiplication in MLWE-768. The PRNG (”PR”) generating $\mathbf{A}$ uses the internal PRNG states stored in the RAM. The modular arithmetic unit (”MA”) computes modular multiplication and addition with the error coefficients supplied by the BN units. The polynomial products are summed up to obtain $\mathbf{c}_1 = \mathbf{A}^\intercal \mathbf{e}_1 + \mathbf{e}_2$.



domain.

A standard matrix-vector product can be recognized in the loops labelled `col` and `row`. The `coeff` loop iterates over the coefficients of the polynomials in matrix $\mathbf{A}$ and $\mathbf{e}_1$. The matrix $\mathbf{A}$ is not read from memory, but computed “on the fly”. The k^2 internal PRNG states are read from memory and the PRNG is used to generate the coefficients of the k^2 polynomials in $\mathbf{A}$. The `reduce` and `reduce_fast` functions perform modular reduction, the `prng` function samples a 13-bit signed integer, and the `DW` macro casts the operands of the multiplication to the `int26` type, to get a 26-bit signed integer as result.

In order to generate a parallel architecture, the directives have to be specified accordingly in Vivado HLS. Our goal is to compute all of the k^2 polynomial multiplications simultaneously. We set the directive `pipeline` on the `coeff` loop. This directive forces all subloops to be completely unrolled. All of the k^2 operations in the `col` and `row` loops will be performed in parallel.

We use the directive `array_partition` to partition arrays `E1[k][n]` and `C1[k][n]` into k different arrays. This will distribute the arrays over k different BRAMs each. Then k values can be loaded from the array `E1` at the same time and k values can be written to `C1` at the same time. We apply the same directive to both dimensions of the $k \times k$ array `Trivium_States`.

To generate an architecture for a different vector length k, we run a script using SageMath that creates a new header file defining k and computes a new set of valid keys. The C source code remains the same and the same directives apply. The files generated by the SageMath script also define all the constants used in the implementation, such as n-th roots of unity and exponents parametrizing the modulus. A simple change of parameters in the script is all that is needed to generate architectures for different values of (n, q, k). The C source code remains unchanged. This means that we can even

Figure 5.10: Matrix-vector multiplication $\mathbf{A}^\intercal \mathbf{e}_1$.

```
coeff: for(i=0; i<N; i++){
    col: for(jj=0; jj<K; jj++){
        c1_coeff = 0;
        row: for(j=0; j<K; j++){
            A_coeff = 0;
            prng(Trivium_States[j][jj], &A_coeff);
            c1_coeff += reduce(DW(A_coeff)*DW(E1[j][i]));
        }
        C1[jj][i] = reduce_fast(c1_coeff);
    }
}
```

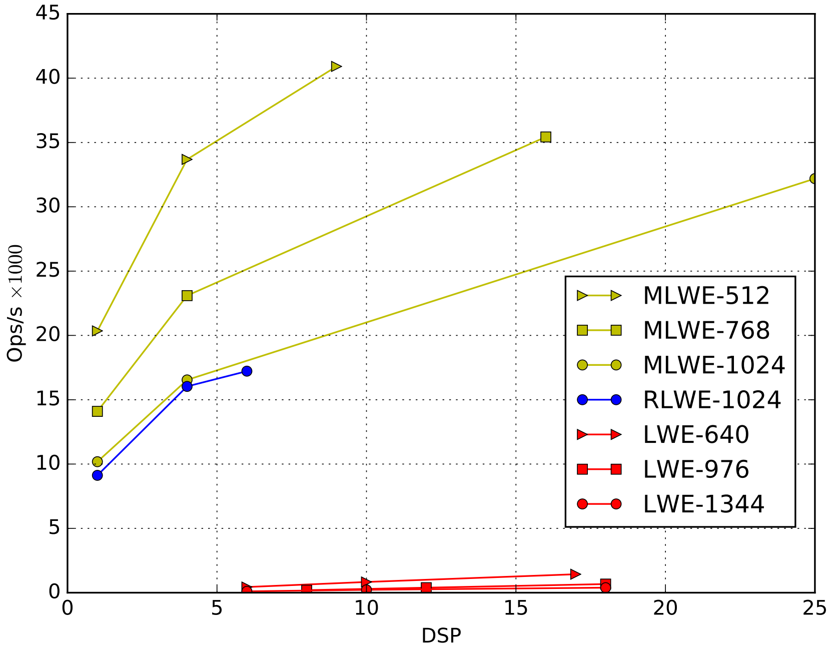
switch between RLWE ($k = 1$) and MLWE implementations ($k = 2, 3$ or 4) by simply generating a new header file. For area optimization we add some specific directives depending on the parameter k. The `allocation` directive for instance, allows to set a limit to the number of DSP blocks in the implementation.

5.5.4 Implementation results

The PRNG is instantiated with the Trivium stream cipher. The results are shown in Table 5.6. In the parallelized implementation of MLWE, the impact of the parameter k on the encryption time is mitigated by adding BRAMs and DSP blocks. The latency (in clock cycles) of the arithmetic part of the scheme is then the same for $k = 2, 3$ and 4. A slight increase in encryption and decryption time is due to the loading and storing of public keys and ciphertexts of increased size. In Table 5.6, increasing k means adding n cycles to the decryption latency, during which the $k \cdot n$ coefficients of the ciphertext part $\mathbf{c}_1$ are loaded. The encryption latency increases by $2n$ cycles since both $\mathbf{b}$ and $\mathbf{c}_1$ consist of $k \cdot n$ coefficients.

The throughput of our LWE, RLWE and MLWE implementations for the different degrees of parallelization discussed, is shown in Figure 5.11. Implementations of RLWE and MLWE using only 1 DSP block and no other optimizations than pipelining are compared to the slightly parallelized (computing NTTs simultaneously) and maximally parallelized implementations. For the RLWE and MLWE implementations, the throughput is increased by computing the NTTs during the encryption in parallel. Further parallelization obtained by unrolling loops increases the throughput even more, up to almost 2× the throughput of the slightly parallelized version in the case of MLWE-1024. The gain in throughput for the RLWE-1024 however, is limited to only 7 percent compared to the slightly parallelized version. This is due to the fact that the memory access patterns of the NTT prevent further parallelization. In RLWE these NTTs consist of 10 stages of 512 butterfly operations each,

Figure 5.11: Throughput (in encryptions per second) *vs* area (in DSP blocks) trade-offs for various parallelism levels. The most left point of each curve corresponds to a sequential architecture, the middle point embeds parallel NTTs (for RLWE/MLWE) and the most right point represents a full parallel architecture.



while in MLWE only 8×128 butterfly operations have to be computed. The potential for parallelization provided by the matrix structure, is clearly an advantage for MLWE compared to RLWE. Even if the number of DSPs is increased, for MLWE-1024 this represents less than 4 percent of the total number of DSPs available on the Artix-7. For the LWE implementations the throughput increases when unrolling the matrix multiplication loop. It remains however, far below those of MLWE and RLWE.

Table 5.6: Parallelized MLWE for different security levels

Size	Freq.	Time (enc/dec)	Area
k	MHz	μ s	DSP, BRAM, Slices, LUT
2	204	25/14	9, 17, 4565, 8584
3	196	29/16	16, 25, 6271, 12383
4	196	32/17	25, 29, 8988, 16803

5.6 Randomness generation and CCA implementations

We discuss the use of rejection sampling for the public key generation, the use of a more secure PRNG and the transformation to CCA secure implementations.

5.6.1 Rejection sampling

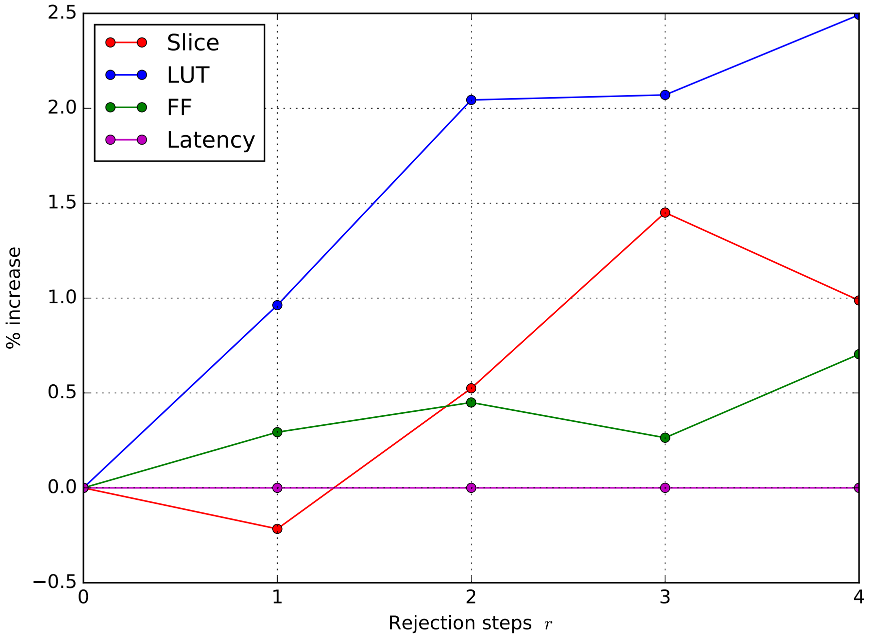
To generate the coefficients of public key $\mathbf{A}$, uniform sampling over $\mathbb{Z}_q$ is needed. The naive way of sampling the uniform distribution over $\mathbb{Z}_q$ is to generate $w = \lceil \log_2(q) \rceil$ random bits defining a w-bit number a and returning $a \bmod q$. This results in a biased distribution: for any $a_0 \in \{0, \ldots, 2^w - q - 1\}$ and $a_1 \in \{2^w - q, \ldots, q - 1\}$, the probability of obtaining a_0 is twice as high as the probability of obtaining a_1. The bias is determined by the probability of obtaining an integer in the range $\{q, \ldots, 2^w - 1\}$, which is equal to $\frac{2^w - q}{2^w} \approx 2^{-4}$ for $q = 7681$. To reduce the bias in the obtained distribution, rejection sampling can be performed. This requires generating a number of random integers $a_0, \ldots, a_r$ and selecting one that is in the interval $[0, q-1]$. The sampling algorithm using r rejection steps, has a probability of returning an integer in the range $\{q, \ldots, 2^w - 1\}$ of approximately $2^{-4(1+r)}$. The impact of the number of rejection steps on the area utilization is shown in Figure 5.12. There is a clear increase in usage of slices, LUTs and flipflops as rejection steps are added. With respect to the total area however, even the additional 189 LUTs (for $r = 4$) represent less than a 4 percent increase. The computation time is largely independent of the number of rejection steps for the range of r considered in Figure 5.12.

5.6.2 Alternative PRNG

While the Trivium PRNG has a good performance, it has a key space of 80 bits, which is less than the number of security bits (128, 192 or 256, depending on the parameter set) targeted by Kyber, NewHope and Frodo. An attacker has no direct access to the PRNG output that is used for error sampling. However, the correctness of a Trivium key guess can be checked by reconstructing $\mathbf{e}_1$ using the PRNG and verifying that $\mathbf{e}_1^\intercal \mathbf{A} \approx \mathbf{c}_1$. The Trivium key can therefore be found in 2^{80} operations. If the Trivium key is compromised, an attacker may compute $\mathbf{c}_2 - \mathbf{b}\mathbf{e}_1 \approx \left\lfloor \frac{q}{2} \right\rfloor \mu$ to recover the message. An exhaustive search in the 80 bit key space could thus be used for message recovery attacks.

In Kyber, NewHope and Frodo it is suggested to use SHAKE-256 or similar algorithms as PRNG. Some schemes propose to use the less secure SHAKE-128 to generate the public key part A. We implement a hybrid version using Trivium for the public key and SHAKE-256 for the error samples, as was done in [64]. We also implement a variant that uses SHAKE-256 for both error sampling and generation of A. The results are shown in figure 5.13. The Keccak implementation, when synthesized separately, takes up 1770 slices, 3782 LUTs and 5121 flipflops, and computes the 24 round Keccak algorithm in 25 clock cycles. The overhead that the SHAKE implementation imposes on the area and timing results is shown in Figures 5.14 and 5.15. The number of DSPs used remains unchanged.

Figure 5.12: Area increase of MLWE-1024 implementation due to rejection sampling.



The hybrid version, using Trivium for the public key, has a clear advantage over the SHAKE-256 only variant in terms of area utilization and computation time. The obtained frequency of the RLWE implementation is heavily impacted by substituting Trivium for SHAKE, dropping from 256 MHz (Trivium) to 166 MHz (SHAKE).

5.6.3 CCA secure implementations

We transform our CPA-only secure LWE, RLWE and MLWE implementations (with hybrid sampling mode) into CCA secure implementations using algorithms 6 and 7. Hash functions H and G are instantiated with the SHA3-256 algorithm. Complete results are shown in Table 5.9. A comparison between the timing results of the CPA-only implementations and the CCA implementations is shown in Figure 5.16. For LWE, the additional hash function evaluations have relatively little impact on the timing results. The computation time difference between CPA-secure and CCA-secure encryption are almost entirely accounted for by the matrix multiplication in the encryption algorithm. For MLWE and RLWE however, this is not the case. The difference between CPA-enc and CCA-enc is due to the hash functions. This is also holds for the difference between CCA-dec and the sum of CPA-enc and CPA-dec. Additional slow-down is caused by a drop in obtained frequency for CCA implementations. The impact on the area is shown in Figure 5.17. There is a clear increase in DSPs for RLWE and MLWE, showing that the sharing of resources is not optimal.

Figure 5.13: Impact of the choice of PRNG on the encryption time.

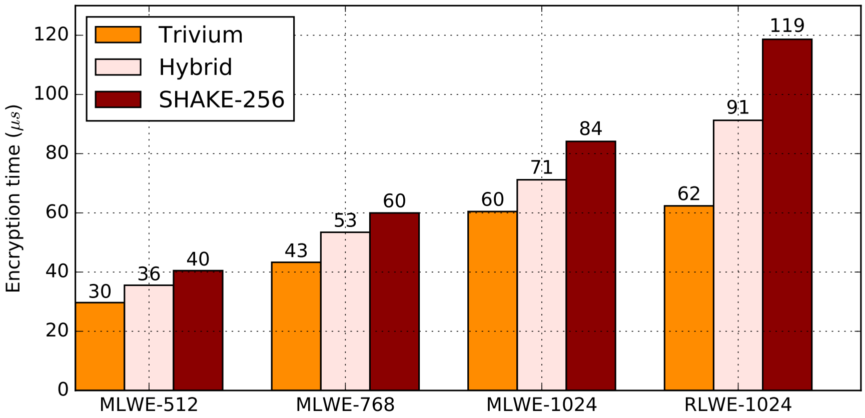


Figure 5.14: Impact of the choice of PRNG on area in slices.

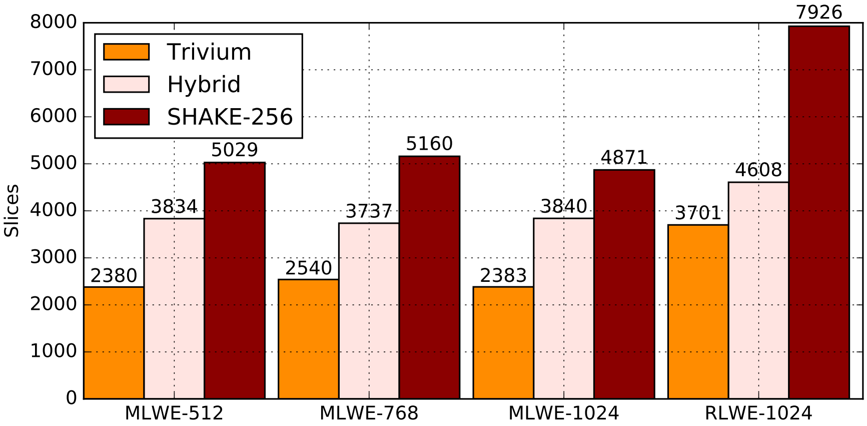


Figure 5.15: Impact of the choice of PRNG on time × area.

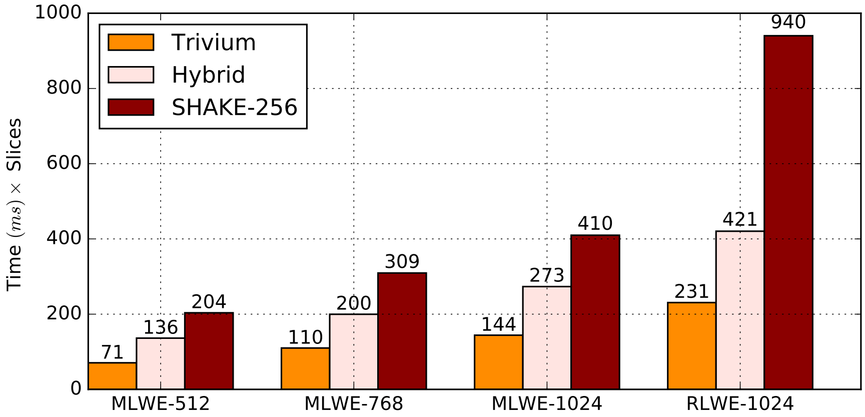


Figure 5.16: Encryption and decryption time of CCA2 and CPA implementations. RLWE and MLWE times are in μs, while ms are used for LWE.

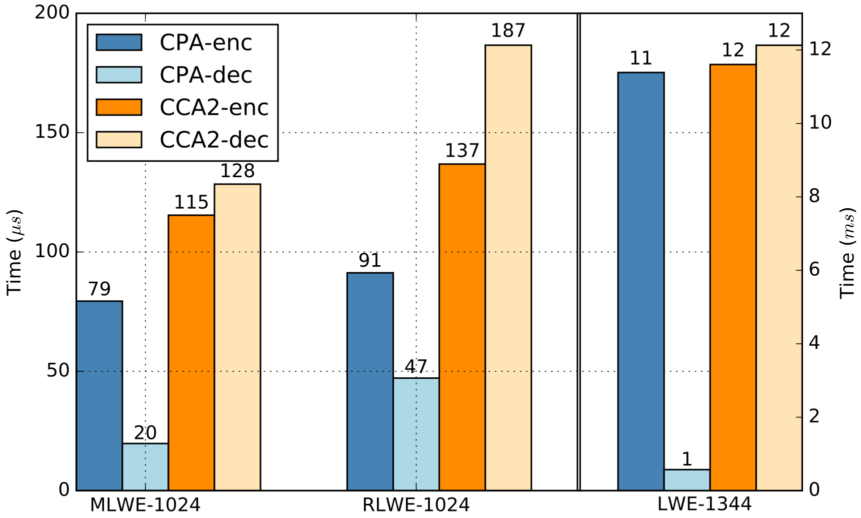
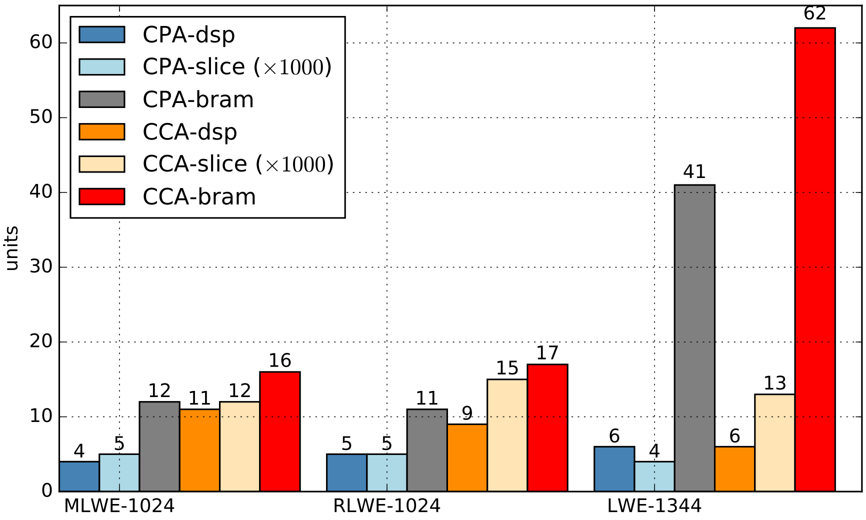


Figure 5.17: Area comparison between CCA2 and CPA only implementations.



5.7 Comparison with other works

A selection of our implementation results of the encryption/encapsulation algorithm is compared to results found in other works in Table 5.7. All of our results and the results from [44] are obtained using HLS. The best MLWE/MLWR implementation from the state of the art is given by [44]. The RLWR and MLWR implementations from [5] and [44] compute the Round5 [11] and Saber [45] encapsulation respectively. The advantage of RLWR is that there is no need to implement modular reduction: integer arithmetic is computed in $\mathbb{Z}_q$ where q is some power of two. There is no need for (binomial) error sampling either: the errors are generated by setting a number of LSBs to zero. The drawback is that the NTT cannot be used for polynomial multiplication for RLWR (and MLWR).

Our CPA-secure implementation of RLWE-1024 is more than 20% faster and at least twice as small as the one from [74]. Note that our result includes area utilization for both encryption and decryption, whereas the results from [74] are only for the implementation of the Server part of a Client-Server-Client key exchange. The two other CPA-only RLWE/RLWR implementations by [90] and [5] have an even higher computation time. Our CCA-secure implementation is outperformed by the very recent result from [115], designed with Verilog HDL and highly optimized.

In [44], the inefficient (compared to the NTT) polynomial multiplication algorithm is compensated by adding slices and LUTs for parallel computation. Their RLWR and MLWR seem faster than our RLWE and MLWE. However, this is largely due to the difference in obtained frequency resulting from the use of different FPGAs. We generate implementations on the same FPGA as [44] (Zynq Ultrascale), and observe that our MLWE-1024 is actually 4% faster than theirs (also see table 5.8), while using 28 times fewer DSPs. The enormous difference in area efficiency is mostly due to the advantage that the NTT has over other multiplication methods. This drawback for RLWR and MLWR schemes like Saber and Round5 seems to significantly outweigh the advantage of not having to compute modular reduction and binomial sampling (needed in Kyber and NewHope). This also explains why the Saber implementation by [87] uses 2.5 times more DSPs than our CCA secure MLWE-768, while our implementation is 47 times faster than theirs.

Our CPA-secure LWE is faster than [64] for similar area utilization, especially for $k = 640$. Our CCA secure LWE also implements decapsulation, which explains the large difference in area results with [64]. Their area results are for encapsulation only, which consists of one encryption and some computations of the hash function. The fastest results for LWE are given by [44].

It should be noted that the choice of FPGA accounts for quite a few differences in performance. To illustrate this, results for our CCA secure MLWE-1024 implementation on other FPGAs are shown in Table 5.8. High end FPGAs run on higher frequencies, reducing the total computation time with a factor of more than 2 for the Zynq UltraScale FPGA, which was also used by [44].

						Freq.	Time	Area
Src.	P	Algorithm	PRNG	Type	FPGA	MHz	μs	DSP, BRAM, Slice, LUT
[74]		RLWE-1024	SHAKE	K-E	xc7z020	131	79	8, 14, n.a. 20826
[90]		RLWE-1024	SHAKE	K-E	xc7a35t	117	1532	2, 4, n.a., 4498
TW	m	RLWE-1024	Trivium	CPA	xc7a200	259	63	4, 10, 3701, 10112 (*)
[44]		RLWR-1170	SHAKE	CCA	xczu9eg	212	30	0, 4, 18733, 91166 (*)
[115]		RLWE-1024	SHAKE	CCA	xc7z020	200	62	2, 8, n.a, 6781 (*)
TW	m	RLWE-1024	Hybrid	CCA	xc7a200	167	137	9, 17, 14026, 42062 (*)
TW	l	LWE-640	Trivium	CPA	xc7a200	200	2275	6, 16, 1629, 4311 (*)
TW	m	LWE-640	Trivium	CPA	xc7a200	208	1201	10, 16, 1692, 4490 (*)
TW	h	LWE-640	Trivium	CPA	xc7a200	213	698	17, 20, 2025, 5360 (*)
[64]	l	LWE-640	Hybrid	CCA	Artix-7	183	4624	4, 0, 1338, 4620
[64]	m	LWE-640	Hybrid	CCA	Artix-7	177	2342	8, 0, 1485, 5155
[64]	h	LWE-640	Hybrid	CCA	Artix-7	171	1212	16, 0, 1692, 5796
[65]		LWE-640	Hybrid	CCA	xc7a35t	167	19608	1, 11, 1855, 6745
[44]		LWE-640	SHAKE	CCA	xczu9eg	402	352	32, 27, 1186, 7213 (*)
TW	l	LWE-640	Hybrid	CCA	xc7a200	159	2972	5, 37, 12951, 39077 (*)
TW	l	LWE-976	Trivium	CPA	xc7a200	200	5123	8, 16, 1601, 4322 (*)
TW	m	LWE-976	Trivium	CPA	xc7a200	213	2577	12, 16, 1738, 4633 (*)
TW	h	LWE-976	Trivium	CPA	xc7a200	208	1493	18, 20, 2307, 5943 (*)
[64]	l	LWE-976	Hybrid	CCA	Artix-7	180	10638	4, 0, 1455, 4996
[64]	m	LWE-976	Hybrid	CCA	Artix-7	175	5464	8, 0, 1608, 5562
[64]	h	LWE-976	Hybrid	CCA	Artix-7	168	2857	16, 0, 1782, 6188
[65]		LWE-976	Hybrid	CCA	xc7a35t	167	45455	1, 16, 1985, 7209
[44]		LWE-976	SHAKE	CCA	xczu9eg	402	760	32, 34, 1190, 7087 (*)
TW	l	LWE-976	Hybrid	CCA	xc7a200	167	6317	14, 37, 13468, 41100 (*)
TW	l	LWE-1344	Trivium	CPA	xc7a200	200	9506	6, 25, 1439, 3832 (*)
TW	m	LWE-1344	Trivium	CPA	xc7a200	222	4491	10, 25, 1559, 3946 (*)
TW	h	LWE-1344	Trivium	CPA	xc7a200	213	2574	18, 29, 1892, 4960 (*)
[44]		LWE-1344	SHAKE	CCA	xczu9eg	417	1328	32, 35, 1215, 7015 (*)
TW	l	LWE-1344	Hybrid	CCA	xc7a200	167	11606	6, 62, 12299, 37342 (*)
TW	m	MLWE-512	Trivium	CPA	xc7a200	257	30	4, 11, 2380, 5538 (*)
TW	h	MLWE-512	Trivium	CPA	xc7a200	204	25	9, 17, 4565, 8584 (*)
[44]		MLWR-512	SHAKE	CCA	xczu9eg	322	43	256, 7, 1989, 12343 (*)
TW	m	MLWE-512	Hybrid	CCA	xc7a200	170	60	11, 16, 11028, 34206 (*)
TW	m	MLWE-768	Trivium	CPA	xc7a200	257	44	4, 11, 2540, 6031 (*)

TW	h	MLWE-768	Trivium	CPA	xc7a200	196	29	16, 25, 6271, 12383 (*)
[44]		MLWR-768	SHAKE	CCA	xczu9eg	322	49	256, 7, 1993, 12566 (*)
TW	m	MLWE-768	Hybrid	CCA	xc7a200	167	88	11, 16, 11890, 34145 (*)
TW	m	MLWE-1024	Trivium	CPA	xc7a200	250	61	4, 11, 2383, 5515 (*)
TW	h	MLWE-1024	Trivium	CPA	xc7a200	196	32	25, 29, 8988, 16803 (*)
[44]		MLWR-1024	SHAKE	CCA	xczu9eg	322	50	256, 7, 2341, 12555 (*)
TW	m	MLWE-1024	Hybrid	CCA	xczu9eg	417	48	9, 16, 9314, 44964 (*)
TW	m	MLWE-1024	Hybrid	CCA	xc7a200	170	116	11, 16, 11567, 33707 (*)

Table 5.7: CPA and CCA-secure Encryption or Encapsulation (CPA, CCA) or 'Server' part in Client-Server-Client key exchange (K-E). The "P" column indicates the level of parallelism (low/medium/high) used. If marked with (*), resource results are for both encryption and decryption. 'Hybrid' in the PRNG column means Trivium + SHAKE.

5.8 Conclusion

We implemented CPA and CCA secure LWE, RLWE and MLWE based cryptosystems on FPGA. To the best of our knowledge, the CCA secure MLWE implementation is a first. Our architectures generated using HLS are up to 28 times smaller in terms of DSPs than other works using HLS [44], while faster than works using VHDL/Verilog (such as [74, 90, 5, 65]). For MLWE in particular, the timing versus area trade-offs of our implementations are better than the cited works, taking in account the differences between FPGAs. We showed how HLS can be used effectively to parallelize implementations. We also evaluated the impact of the choice of the PRNG on the performance of the encryption. Using Trivium instead of SHAKE to generate the pseudorandom part of the public key, the encryption can be accelerated. Even more speed-up is obtained when using Trivium for the error sampling as well, although this decreases the theoretical security of the scheme.

Table 5.8: CCA-secure MLWE-1024 using SHAKE-256 for error sampling for different FPGAs, using Vivado version 2018.3.

	Freq.	Time (enc/dec)	Area
FPGA	MHz	μs	DSP, BRAM, Slices, LUT
xc7a100t	200	99/110	11, 16, 11322, 35607
xc7k325t	286	70/77	9, 16, 12066, 34175
xc7v585t	286	70/77	9, 16, 12508, 35718
xczu9eg	417	48/53	9, 16, 9314, 44964
xcku040	333	61/68	9, 16, 7238 43101,
xcvu080	286	69/77	9, 16, 6474 33979,

Table 5.9: CCA-secure implementations using SHAKE-256 for error sampling.

	Freq.	Time (enc/dec)	Area
Algorithm	MHz	μs	DSP, BRAM, Slices, LUT
LWE-640	158	2972/3234	5, 37, 12951, 39077
LWE-976	166	6317/6698	14, 37, 13468, 41100
LWE-1344	166	11606/12130	6, 62, 12299, 37342
RLWE-1024	166	137/187	9, 17, 14026, 42062
MLWE-512	169	60/70	11, 16, 11028, 34206
MLWE-768	166	88/100	11, 16, 11890, 34145
MLWE-1024	169	116/129	11, 16, 11567, 33707

Table 5.10: CPA-secure encryption and decryption using Trivium PRNG, including results from Tables 5.2, 5.4, 5.5 and 5.6.

Algo.	MHz	Enc/dec(μs)	DSP, BRAM, Slices, LUT
RLWE-1024	206	110/47	1, 11, 3820, 10563
RLWE-1024	258	63/38	4, 10, 3701, 10112
RLWE-1024	251	59/35	6, 16, 4474, 12301
MLWE-512	256	30/12	4, 11, 2380, 5538
MLWE-768	256	44/15	4, 11, 2540, 6031
MLWE-1024	250	61/17	4, 11, 2383, 5515
MLWE-512	204	25/14	9, 17, 4565, 8584
MLWE-768	196	29/16	16, 25, 6271, 12383
MLWE-1024	196	32/17	25, 29, 8988, 16803
LWE-640	200	2275/232	6, 16, 1629, 4311
LWE-976	200	5123/353	8, 16, 1601, 4322
LWE-1344	200	9506/486	6, 25, 1439, 3832
LWE-640	208	1201/223	10, 16, 1692, 4490
LWE-976	212	2577/333	12, 16, 1738, 4633
LWE-1344	222	4491/438	10, 25, 1559, 3946
LWE-640	212	698/219	17, 20, 2025, 5360
LWE-976	208	1493/340	18, 20, 2307, 5943
LWE-1344	212	2574/457	18, 29, 1892, 4960

Chapter 6

Countermeasures against Side-Channel Attacks

6.1 Introduction

This chapter is based on our work [117] published and presented at IndoCrypt 2019. In this chapter, we discuss side-channel vulnerabilities in the decryption algorithm of the RLWE based cryptosystem. We identify the vulnerability of certain operations in the decryption algorithm by simulating SCA in the Hamming weight model. To protect these operations, we implement countermeasures from the state of the art, improve these countermeasures, and propose new protections. All countermeasures are implemented on FPGA to measure and compare the impact on the computation time and area utilization of each countermeasure.

6.1.1 Correlation Power Attack simulations in Python

The polynomial multiplication in the NTT domain $\mathbf{c}_1 \odot \mathbf{s}$ consists of n independent multiplications in the field $\mathbb{Z}/q\mathbb{Z}$. They are of the form $s \cdot c \bmod q$, where s is a coefficient of the secret key and c is a coefficient of the input ciphertext. We simulate DPA attacks on **one single modular multiplication** of a known input coefficient c with an unknown secret key coefficient s.

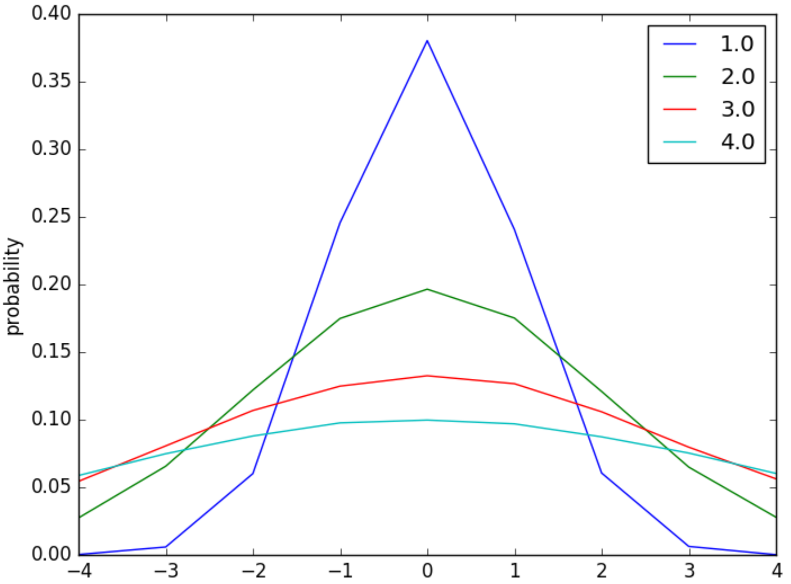
In our attack model, for each multiplication $c \cdot s \bmod q$ the attacker knows the value of c and learns an approximation of the Hamming weight of the result:

$$HW(c \cdot s \bmod q) + \mathcal{N}(0, \sigma), \tag{6.1}$$

where $\mathcal{N}(0, \sigma)$ is the rounded Gaussian distribution with standard deviation σ (see Figure 6.1). The attacker proceeds as follows:

1. Generate a number N_c of random values $r_1, \ldots, r_{N_c}$ in $\mathbb{Z}/q\mathbb{Z}$ (these are the first coefficients of

Figure 6.1: Rounded Gaussian distributions for $\sigma \in \{1, 2, 3, 4\}$.



N_c random ciphertexts).

2. For each subkey candidate $\tilde{s} \in \mathbb{Z}/q\mathbb{Z}$, compute the predictions

$$P(\tilde{s}) := \Big(HW(\tilde{s} \cdot r_1 \bmod q), \ldots, HW(\tilde{s} \cdot r_{N_c} \bmod q)\Big).$$

3. Observe the approximations:

$$Q := \Big(HW(s \cdot r_1 \bmod q) + \mathcal{N}(0, \sigma), \ldots, HW(s \cdot r_{N_c} \bmod q) + \mathcal{N}(0, \sigma)\Big).$$

4. Compute the Pearson correlation coefficients

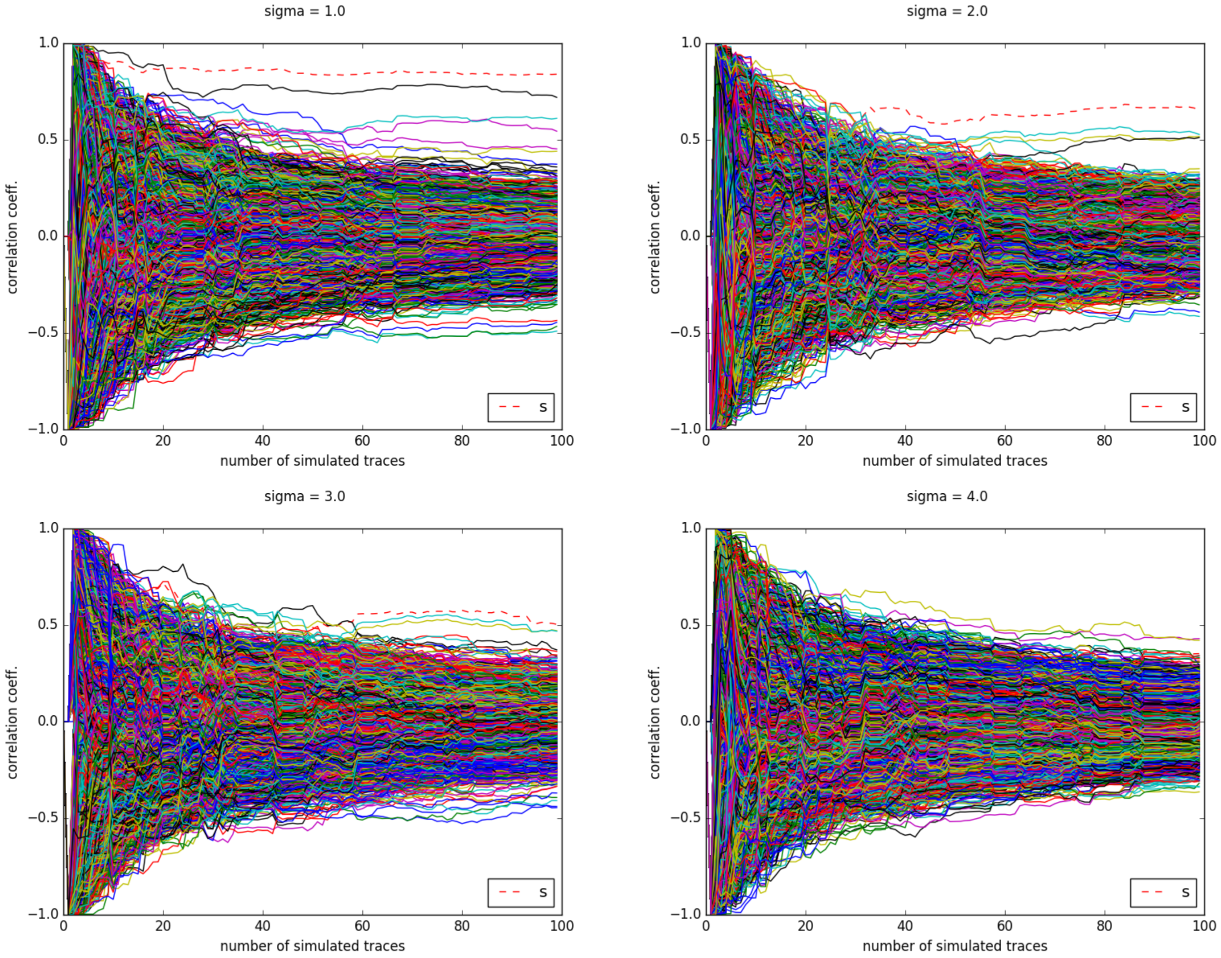
$$\rho_{\tilde{s}} = PCC(Q, P(\tilde{s}))$$

for all $\tilde{s}$.

5. Find the $\tilde{s}$ for which $\rho_{\tilde{s}}$ is maximum.

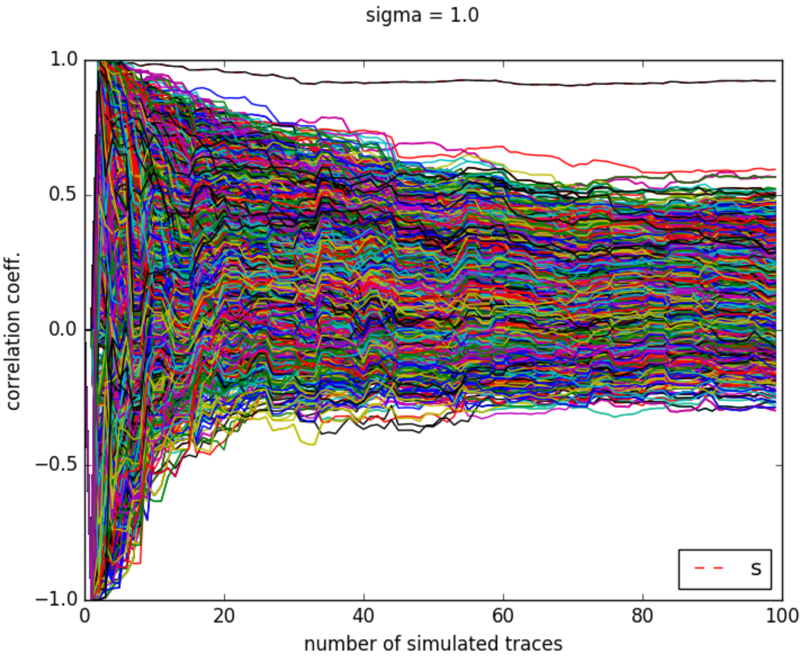
The standard deviation σ is a measure for the noise. Simulations results are shown in Figure 6.2. For $\sigma \leq 3$, the correlation between the correct key coefficient guess and the simulated traces is

Figure 6.2: Simulations of correlation power attacks on one single modular multiplication, for 4 different degrees of noise: $\sigma \in \{1, 2, 3, 4\}$. For each subkey guess, the correlation with the "observed" (simulated) traces is drawn as a function of the number of simulated traces. The red dotted line is the correct subkey guess.



maximum after obtaining a sufficient amount of traces. For $\sigma = 1$, this amount is equal to 20, for $\sigma = 2$ at least 35 traces are needed and for $\sigma = 3$ the attacker needs a minimum of 60 traces. The other subkey guesses that yield high correlation with the "observed" traces, are the guesses $2s$, $4s$ and $8s$, where s is the correct subkey guess. The reason for this is that a multiplicaton by 2 has no effect on the Hamming weight of the product in $\mathbb{Z}$, and relatively little effect when the product is computed in $\mathbb{Z}_q$. For $\sigma = 4$, it seems to be getting impossible to distinguish between these false positives and the correct subkey guess within 100 traces. This (near to) linearity makes that the modular multiplication is not a perfect target for power analysis, as opposed to for example the completely non-linear Sbox in the AES algorithm.

Figure 6.3: The "black" line with the highest correlation actually consists of 4 overlapping lines corresponding to the subkey guesses $s, 2s, 4s$ and $8s$.



To illustrate this further, let us consider to attack model in which the attacker learns an approximation of the Hamming weight of the **unreduced** result: $HW(c \cdot s) + \mathcal{N}(0, \sigma)$. As can be seen in Figure 6.3, this method yields multiple false positives, even for the very small noise parameter $\sigma = 1$. For any number of traces (smaller than 100), the correlation of these false positives is exactly equal to that of the correct subkey guess. The problem is that in $\mathbb{Z}$ the Hamming weight of $2^i s \cdot c$ is constant for all $i \in \mathbb{N}$.

The simulations discussed in this section show that the secret key coefficients can be recovered using correlation power attacks in the defined attack model. While in the simulations we targeted the point-wise multiplication of the secret key with the ciphertext input, other operations may be just as vulnerable. The attack model used in the simulations assumed that the attacker is able to obtain an approximation of the Hamming weight of the modular multiplication result at the moment that it is written to the registers. Key dependent data is written to registers throughout the execution of the decryption algorithm. The decryption algorithm decodes the coefficients of some polynomial $\mathbf{d}$ where $\mathbf{d}$ is computed as $\mathbf{d} = \mathbf{c}_2 - \mathrm{NTT}^{-1}(\mathbf{c}_1 \odot \mathbf{s})$. It should be noted that knowledge of the coefficients of $\mathbf{d}$ leads to complete key recovery in the chosen plaintext attack model. Since $\mathbf{c}_1$ and $\mathbf{c}_2$ are known inputs and $\mathbf{c}_1$ is invertible in $\mathcal{R}_q$ with high probability, one can compute $\mathbf{c}_1^{-1} \cdot (\mathbf{c}_2 - \mathbf{d}) = \mathbf{s}$. To prevent SCAs on the coefficients of $\mathbf{d}$, these coefficients should not be computed directly. Instead, a randomized version of $\mathbf{d}$ should be used. The last step, which consists of decoding the coefficients, then takes a randomized version of $\mathbf{d}$ as input. The decoder should therefore be able to decode randomized inputs.

6.2 Countermeasures in the State of the Art

We now present the main countermeasures from literature against power analysis attacks on RLWE.

Masking

In [102] the secret key is split in two shares: $\mathbf{s} = \mathbf{s}' + \mathbf{s}''$ for some uniformly random $\mathbf{s}'$ at the start of each decryption. The linear part of the decryption function is computed twice: first the ciphertext is decrypted (but not decoded) using secret key $\mathbf{s}'$ and then using secret key $\mathbf{s}''$, yielding two polynomials $\mathbf{d}'$ and $\mathbf{d}''$, as shown in Algorithm 14.

Algorithm 14 Masked RLWE decryption [102]

```
    Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
    Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $\mathbf{s}' \xleftarrow{\$} \mathcal{R}_q$
    $\mathbf{s}'' \leftarrow \mathbf{s} - \mathbf{s}'$
    $\mathbf{d}' \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}'$
    $\mathbf{d}'' \leftarrow -\mathbf{c}_1\mathbf{s}''$
    for $0 \le i < n$ do
        $d_i \leftarrow$ Decode_masked($d_i', d_i''$)
```

The final step consists of decoding the coefficients of $\mathbf{d} = \mathbf{d}' + \mathbf{d}''$ to bits. This is a non linear operation, that is, $\textsc{Decode}(a+b)$ is not necessarily equal to $\textsc{Decode}(a)+\textsc{Decode}(b)$. As an example, if $a = b = \lfloor \frac{q}{6} \rfloor$, then $2 \times \textsc{Decode}\left(\lfloor \frac{q}{6} \rfloor\right) = 0$ but $\textsc{Decode}\left(2 \times \lfloor \frac{q}{6} \rfloor\right) = 1$. This means that one cannot simply apply the decoder to the coefficients of $\mathbf{d}'$ and $\mathbf{d}''$ separately and then add the results in $\mathbb{Z}_2$ to obtain the correct plaintext. The non-linearity of the decoding algorithm is illustrated in Figure 6.4.

Because of the DPA scenario mentioned above, the two shares $\mathbf{d}'$ and $\mathbf{d}''$ should not be recombined before decoding to bits. Let d' and d'' denote a coefficient (of some fixed index) of polynomials $\mathbf{d}'$ and $\mathbf{d}''$ respectively. A *masked decoder* takes as input two coefficients $(d', d'') \in \mathbb{Z}_q^2$ and computes the value of $\textsc{Decode}(d' + d'')$ **without computing** $d' + d''$. The solution from [102] makes use of the fact that for some $(d', d'') \in \mathbb{Z}_q^2$ it is easy to deduce the value of $\textsc{Decode}(d' + d'')$. For instance, if $0 \le d' < \frac{q}{4}$ and $\frac{q}{4} \le d'' < \frac{q}{2}$ then it must hold that $\frac{q}{4} \le (d' + d'') < \frac{3q}{4}$, therefore the coefficient decodes to 1. Similar "easy cases" exist, but not all $(d', d'') \in \mathbb{Z}_q^2$ can be resolved in this way. If both d' and d'' lie between 0 and $\frac{q}{4}$, all we know is that $0 \le (d' + d'') < \frac{q}{2}$ and this can decode to either 0 or 1. Examples of hard cases are illustrated in Figure 6.5.

The idea from [102] to solve the hard cases is to *reshare* the two shares: for any $\delta \in \mathbb{Z}_q$ one has $d' + \delta + d'' - \delta = d' + d'' = d$. It is therefore possible to add any constant to one of the shares and subtract the same constant from the other one. This refreshing of the shares is illustrated in Figure 6.6. However, there is no guarantee that $(d' + \delta, d'' - \delta)$ is an easy case. If the new shares

Figure 6.4: The two shares d' and d'' both lie in the right half of $\mathbb{Z}_q$, that is, both decode to 0 if decoded individually. Their sum d on the other hand, decodes to 1. In this case: $\textsc{Decode}(d') + \textsc{Decode}(d'') \neq \textsc{Decode}(d' + d'')$.

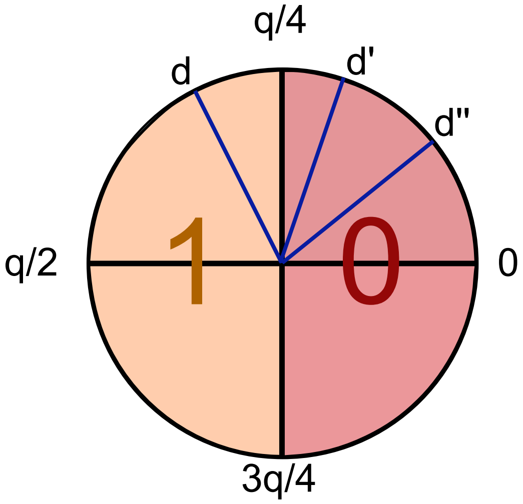


Figure 6.5: Both shares lie in the interval $\{0, \ldots, \frac{q}{4}\}$. Then their sum can lie in either the left or the right half of $\mathbb{Z}_q$.

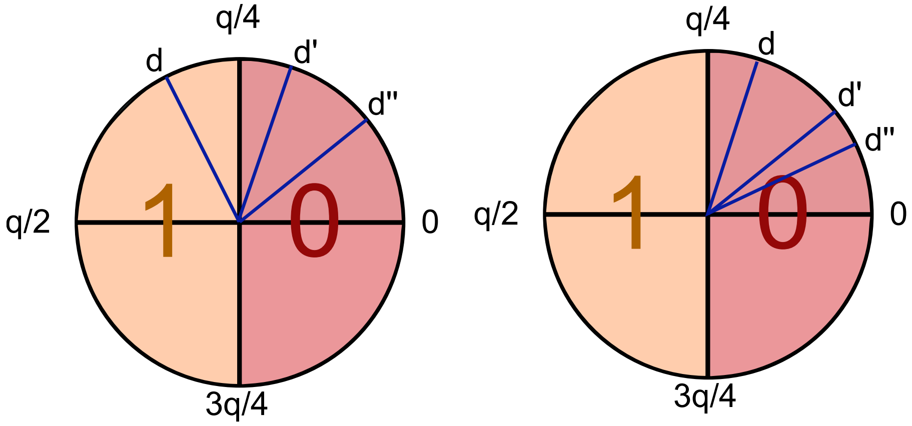
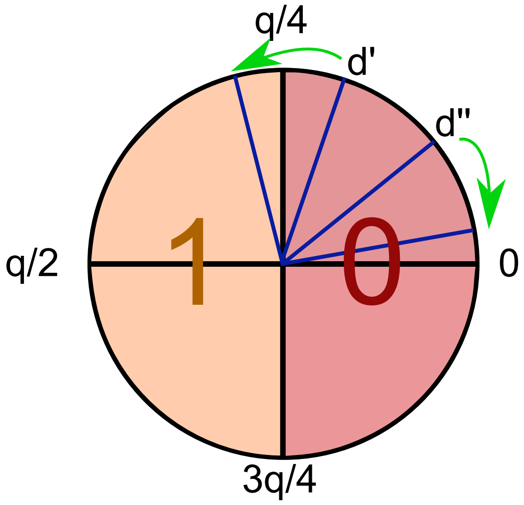


Figure 6.6: Refreshing the two shares d' and d'' yields an easy case, while preserving the correctness of the decoding outcome.



still do not form an easy case, the shares have to be reshared again. In [102] a list of constants $\{\delta_1, \ldots, \delta_{16}\}$ is presented that is supposed to minimize the number of resharings to be performed. Their implementation refreshes the shares 16 times such that with high probability an easy case is obtained in at least one of the 16 iterations.

The computation time overhead due to the 16 iterations and the additional decoding failures are important drawbacks to this solution. [91] propose an alternative masked decoding. Their method effectively decodes without additional decoding failures. The comparison that they make between this decoder and their re-implementation of the one from [102] however shows only a very limited improvement in terms of performance. Their masked decryption takes over 3 times more cycles to compute than the unmasked version. The same implementation also uses a blinding countermeasure proposed by [106].

Blinding

With the blinding countermeasure [106] the polynomials $\mathbf{s}$ and $\mathbf{c}_1$ are multiplied by scalars a and b in $\mathbb{Z}_q$ respectively. The blinded polynomial multiplication is then computed: $(a\mathbf{s}) \cdot (b\mathbf{c}_1) = (ab)\mathbf{s} \cdot \mathbf{c}_1$. The inverse $(ab)^{-1}$ should be computed to obtain $\mathbf{s} \cdot \mathbf{c}_1$. [106] suggested to use (pre-computed) powers of ω and ω^{-1} as blinding factors to avoid the modular inversion. The decoding process cannot be protected from DPA with the scalar blinding method. The blinding multiplication has to be inverted before the coefficients can be decoded (see Algorithm 15).

Algorithm 15 RLWE decryption using the blinding countermeasure [106]

```
Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function DEC($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $i, j \xleftarrow{\$} \{0, \ldots, n-1\}$
    $\mathbf{s} \leftarrow \omega^i \mathbf{s}$
    $\mathbf{c}_1 \leftarrow \omega^j \mathbf{c}_1$
    $\mathbf{d} \leftarrow \mathbf{c}_1 \mathbf{s}$
    $\mathbf{d} \leftarrow \omega^{-(i+j)} \mathbf{d}$
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{d}$
    for $0 \leq k < n$ do
        $d_k \leftarrow$ DECODE($d_k$)
```

Shifting

It is also suggested in [106] to apply a random anti-cyclic shift to the coefficients vector of the polynomials before multiplying. Due to the ring structure, this anti-cyclic shift corresponds to a multiplication by some power of x. For some random $i, j < n$, $(x^j \mathbf{s}(x)) \cdot (x^i \mathbf{c}_1(x)) = x^{i+j} \mathbf{s}(x) \mathbf{c}_1(x)$ is computed and $\mathbf{s}(x)\mathbf{c}_1(x)$ can be recovered by inversing the shift.

Algorithm 16 RLWE decryption using the shifting countermeasure [106]

```
Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function DEC($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $i, j \xleftarrow{\$} \{0, \ldots, n-1\}$
    $\mathbf{s} \leftarrow x^i \mathbf{s}$
    $\mathbf{c}_1 \leftarrow x^j \mathbf{c}_1$
    $\mathbf{d} \leftarrow \mathbf{c}_2 - x^{-(i+j)} \mathbf{c}_1 \mathbf{s}$
    for $0 \leq k < n$ do
        $d_k \leftarrow$ DECODE($d_k$)
```

Shuffling

Masking, blinding and shifting offer little to no protection against single trace attacks. The single trace attack by [99] exploits leakage from the operations performed during the NTT. In that paper, it is suggested to counter the attack by randomizing the order in which the butterfly operations are computed. During each stage, the $\frac{n}{2}$ butterfly operations can be computed in a random order. The same shuffling methods can also be applied to all the pointwise operations in the decryption.

6.3 New Variants of State of the Art Protections

The protections proposed in this section and the next one are implemented by modifying our base architecture from Figure 5.7 for $n = 256$ and $q = 7681$. In real-world applications these protections should be part of an architecture implementing the CCA2-secure version of the scheme, including a re-encryption of the decrypted ciphertext and several evaluations of some hash function.

6.3.1 Masking with a New Masked Decoder

We implement a variant of the masking scheme described in the state of the art [102], improving the masked decoding process. We propose a simple masked decoder that does not need 16 iterations and that does not increase the decoding failure probability. Let $d', d'' \in \left[0, \frac{q}{4}\right)$, then $d' + d'' \in \left[0, \frac{q}{2}\right)$. If either d' or d'' were to be shifted by exactly the right amount (see Figure 6.7), then we would be able to determine if either $d' + d'' \in \left(\frac{-q}{4}, \frac{q}{4}\right)$ or $d' + d'' \in \left(\frac{q}{4}, \frac{3q}{4}\right)$. The trick is then to find a $\delta \in \left[\frac{-q}{4}, \frac{q}{4}\right]$ such that $d' + \delta$ changes quadrant while $d'' - \delta$ does not (or the other way around). Suppose that $d' \geq d''$ and let $\delta = 1 + \min(\left\lfloor \frac{q}{4} \right\rfloor - d', d'')$. Then, depending on the value of δ, there are two possibilities for the new shares $d' + \delta$ and $d'' - \delta$:

1. $d' + \delta = \left\lfloor \frac{q}{4} \right\rfloor + 1 \in \left[\frac{q}{4}, \frac{q}{2}\right)$ and $d'' - \delta$ is in the same interval as d''. Then $d' + d''$ must be in the interval $\left(\frac{q}{4}, \frac{3q}{4}\right)$, therefore we decode to 1.
2. $d' + \delta$ is in the same interval as d' and $d'' - \delta = -1 \in \left[\frac{-q}{4}, 0\right)$. Then $d' + d''$ must be in the interval $\left[0, \frac{q}{4}\right) \cup \left(\frac{-q}{4}, 0\right]$ and therefore we decode to 0.

Similar solutions can be found for the other hard cases. Let Q_i denote the interval $\left[\frac{iq}{4}, \frac{(i+1)q}{4}\right)$ for $0 \leq i \leq 3$, that we will refer to as "quadrants". The property that allows to solve the easy cases is the following:

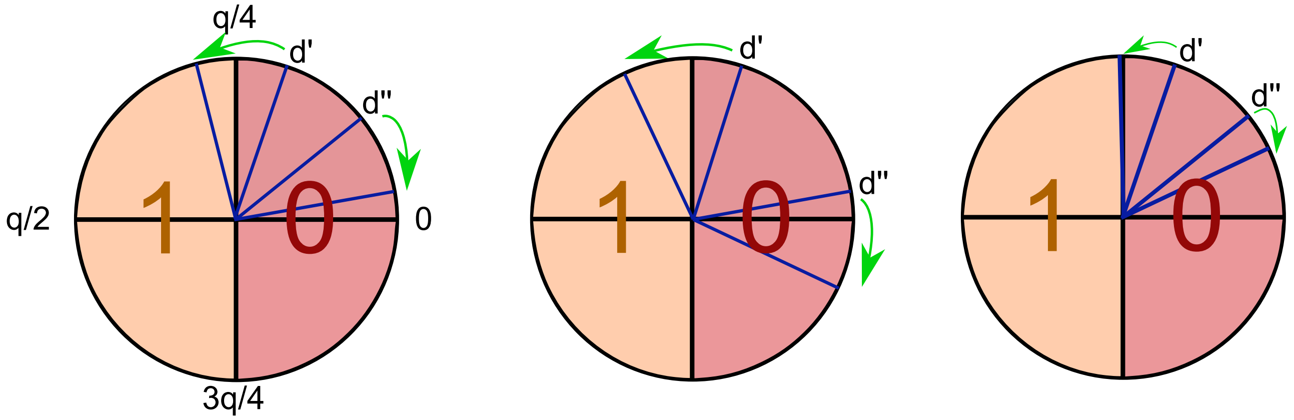
Property 6.3.1. *If $d' \in Q_i$ and $d'' \in Q_j$ then $(d' + d'') \in Q_{i+j \bmod 4} \cup Q_{i+j+1 \bmod 4}$.*

In the remainder of this section, we let i and j be the quadrant indices of d' and d'' respectively. For $i + j = 1 \bmod 4$ it follows from Property 1 that $(d' + d'') \in Q_1 \cup Q_2$. In other words, the sum lies in the left half of $\mathbb{Z}_q$ and therefore decodes to 1. Similarly, the $(d', d'') \in \mathbb{Z}_q^2$ for which $i + j = 3 \bmod 4$ are easy cases and decode to 0.

The hard cases are given by $(d', d'') \in \mathbb{Z}_q^2$ for which $i + j = 0 \bmod 4$ or $i + j = 2 \bmod 4$, that is, $(d' + d'') \in Q_0 \cup Q_1$ or $(d' + d'') \in Q_2 \cup Q_3$ respectively. To reduce to an easy case, it suffices to move either (but not both) d' or d'' to an adjacent quadrant. Then for the new pair $(d' + \delta, d'' - \delta)$ exactly $1 \bmod 4$ is added to or subtracted from the sum $i + j$. Then for the updated i, j it holds that $i + j = 1 \bmod 4$ or $i + j = 3 \bmod 4$ and Property 1 applies.

It is always possible to modify the sum $i + j$ for the i, j corresponding to the shares by exactly 1. Assume w.l.o.g. that $d' \geq d''$. If $d' \in Q_i$, $d'' \in Q_j$ and d' is closer to $\frac{iq}{4}$ than d'' is to $\frac{(j-1)q}{4}$, then there

Figure 6.7: Left: given that $d', d'' \in \left[0, \frac{q}{4}\right)$, there is no simple way to determine if $d' + d'' \in \left(\frac{q}{4}, \frac{3q}{4}\right)$, *i.e.* this is a hard case. Adding some δ to d' while subtracting the same δ from d'' yields a new pair (d', d'') which is an easy case. Middle: for some hard cases, adding and subtracting a constant δ does not give a solution: the new pair (d', d'') is another hard case. Right: d' is closer to $\frac{q}{4}$ than d'' is to 0. We therefore let $\delta = 1 + \left\lfloor \frac{q}{4} \right\rfloor - d'$. Then by construction, $d' + \delta$ changes quadrant while $d'' - \delta$ does not. It follows that $d = d' + \delta + d'' - \delta > \frac{q}{4}$ and $d < \frac{3q}{4}$. Therefore (d', d'') decodes to 1.



is a δ such that $d' + \delta \in Q_{i+1}$ and $d'' - \delta \in Q_j$. If the opposite holds, then d'' can be moved to Q_{j-1} by subtracting a δ while d' stays in Q_i. The new pair $(d' + \delta, d'' - \delta)$ forms an easy case. This method does not work when the distance δ'' between d'' and $\frac{(j-1)q}{4}$ is equal to the distance δ' between d' and $\frac{iq}{4}$. However, these are exactly the cases for which $d' + d''$ is equal to either $\left\lfloor \frac{q}{4} \right\rfloor$ or $\left\lfloor \frac{3q}{4} \right\rfloor$. This means that even an unmasked decoder would not be able to decode these cases correctly. The parameters in LWE-based cryptosystems are usually chosen such that such cases appear with negligible probability.

The comparison operation $\delta' < \delta''$ has to be implemented with caution. Generally, comparisons are performed by checking the bit sign of the subtraction of its operands. Since $\delta' - \delta'' = -(d' + d'') + \left\lfloor \frac{kq}{4} \right\rfloor$ for some integer $k < 4$, this operation leaks information about the unmasked value of d.
Instead of implementing a combinatory circuit, we have implemented successive accesses to a look-up table to perform the comparison. The implemented algorithm is described in Algorithm 17, where the bits of a and b are denoted $(a_0, \ldots, a_{w-1})$ and $(b_0, \ldots, b_{w-1})$ respectively. The look-up table implements the function T defined by $T(a_i, b_i, Y) = \big(a_i \wedge (\overline{b_i} \vee Y)\big) \vee (\overline{b_i} \wedge Y)$.

Algorithm 17 Returns *True* if and only if $a > b$

```
function COMPARE($a, b$)
    $Y \leftarrow False$
    for $i = 0$ to $w - 1$ do
        $Y \leftarrow T(a_i, b_i, Y)$
    return $Y$
```

Note that it is not necessary to assume that $d' > d''$. Given (d', d'') and their corresponding quadrant indices $(i,j) = index(d',d'')$, the distances $\delta' = \left\lfloor \frac{(i+1)q}{4} \right\rfloor - d'$ and $\delta'' = d'' - \left\lfloor \frac{jq}{4} \right\rfloor$ are computed and compared. We have that:

$$\begin{aligned} \delta' < \delta'' &\iff \left\lfloor \frac{(i+1)q}{4} \right\rfloor - d' < d'' - \left\lfloor \frac{jq}{4} \right\rfloor \\ &\iff \left\lfloor \frac{(j+1)q}{4} \right\rfloor - d'' < d' - \left\lfloor \frac{iq}{4} \right\rfloor, \end{aligned}$$

which means that swapping d' and d'' (and their corresponding indices) does not change the boolean outcome of the comparison. The complete masked decoder is given by Algorithm 18. The new reshared parts $d' + \delta$ and $d'' - \delta$ do not need to be computed explicitely. The comparison of δ' and δ'' yields sufficient information to update the indices (i,j) and determine the decoded bit.

Algorithm 18 Proposed masked decoder for (d',d'')

```
function DECODE($(d',d'')$)
    $r \xleftarrow{\$} \{0,1\}$                                   ▷ Mask for output
    $(i,j) \leftarrow$ index$(d',d'')$                            ▷ Quadrant indices
    if $i+j \equiv 1 \bmod 4$ then
        return $(r,\bar{r})$                                      ▷ Easy cases $i+j \equiv 1$ or 3.
    else if $i+j \equiv 3 \bmod 4$ then
        return $(r,r)$
    else
        $\delta' \leftarrow \left\lfloor \frac{(i+1)q}{4} \right\rfloor - d'$     ▷ Distance to interval boundaries
        $\delta'' \leftarrow d'' - \left\lfloor \frac{jq}{4} \right\rfloor$
        if COMPARE$(\delta'',\delta')$ then
            if $i+j+1 \equiv 1 \bmod 4$ then
                return $(r,\bar{r})$
            else
                return $(r,r)$
        else
            if $i+j-1 \equiv 1 \bmod 4$ then
                return $(r,\bar{r})$
            else
                return $(r,r)$
```

In order to make this masked decoder compatible with CCA2-secure implementations, the output is also masked. Instead of returning the plaintext bit, a random bit is generated and XORed with the unmasked decoding result. The decoder returns both the mask and the masked value.

A total of $2^{n \log q} = 2^{3328}$ different masks can be obtained. The security of the masking scheme with its original decoder is experimentally evaluated by [102]. They also mention the (small) possibility of horizontal DPA attacks targeting the 16 iterations of their masked decoder. Our proposed decoder does not have this vulnerability since it does not use 16 iterations.

6.3.2 Modified Shifting

In practice it is not possible to obtain $x^i\mathbf{s}$ and $x^j\mathbf{c}_1$ by applying anti-cyclic shifts to their coefficients vectors, because they are represented in the NTT domain. To multiply by x^i in the NTT domain, observe that

$$\mathrm{NTT}(x^i) = (1, \omega^i, \omega^{2i}, \dots, \omega^{(n-1)i}), \tag{6.2}$$

and $\mathrm{NTT}((\phi x)^i) = \phi^i \cdot \mathrm{NTT}(x^i)$. All of the coefficients of $\mathrm{NTT}(x^i)$ are already pre-computed, since they are exactly the n powers of ω. Multiplication by x^i can thus be done by a pointwise multiplication with the powers of ω and ϕ^i (for the NWC). This multiplication has to be performed in bit-reversed order, since $\mathbf{s}$ and $\mathbf{c}_1$ are in the NTT domain. In [106] there is no mention of any masked decoder. To secure the complete decryption function, we propose to apply the (normal) decoder to the shifted polynomial $x^{i+j}\mathbf{c}_2(x) - x^{i+j}\mathbf{s}(x)\mathbf{c}_1(x)$, meaning that $\mathbf{c}_2(x)$ should be shifted separately. The plaintext can then be obtained by applying the inverse shift to the decoded polynomial. The minus sign that comes with the anti-cyclic shift does not change the value of the decoded coefficient, because $\forall a \in \mathbb{Z}/q\mathbb{Z}$ it holds that $\textsc{Decode}(a) = \textsc{Decode}(-a)$. The decryption procedure for a ciphertext $(\mathbf{c}_1, \mathbf{c}_2)$ can then be described as in Algorithm 19.

Algorithm 19 RLWE decryption using the shifting countermeasure [106]

```
    Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
    Output: Plaintext $\mathbf{d} = (d_0, \dots, d_{n-1}) \in \{0,1\}^n$
function Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $i, j \xleftarrow{\$} \{0, \dots, n\}$
    $\mathbf{s} \leftarrow \phi^i \mathrm{NTT}(x^i) \odot \mathbf{s}$
    $\mathbf{c}_1 \leftarrow \phi^j \mathrm{NTT}(x^j) \odot \mathbf{c}_1$
    $\mathbf{c}_2 \leftarrow \phi^{i+j} \mathrm{NTT}(x^{i+j}) \odot \mathbf{c}_2$
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$
    for $0 \le k < n$ do
        $d_{k+i+j} \leftarrow \textsc{Decode}(d_k)$
```

6.3.3 Blinding

The blinding countermeasure is implemented by generating two random indices $0 \le i, j < n$ and multiplying $\mathbf{c}_1$ and $\mathbf{s}$ by ω^i and ω^j respectively, as described in Algorithm 15 from the state of the art.

6.3.4 Shifting and Blinding Combined

Both shifting and blinding involve multiplication by the powers of ω and ϕ. To shift the polynomial $\mathbf{s}(x)$ by $i < n$ positions, we compute $\phi^i \cdot \mathrm{NTT}(x^i) \odot \mathbf{s}(x)$. With almost no additional costs, this operation can be combined with the blinding operation by simply modifying the exponents of ω. To shift the polynomial by i positions and blind using ω^{-j} for some $j < n$, we use:

$$\omega^{-j}\phi^i \cdot \mathrm{NTT}(x^i) \odot \mathbf{s}(x) = (\phi^i\omega^{-j}, \phi^i\omega^{i-j}, \ldots, \phi^i\omega^{(n-1)i-j}) \odot \mathbf{s}(x) \tag{6.3}$$

Both $\mathbf{s}$ and $\mathbf{c}_1$ are shifted and blinded (see algorithm 20). The combined blinding factor has to be removed before the decoding. The combination of the two countermeasures is therefore somewhat more expensive than shifting alone. The decoding is performed in the shifted order.

Both shifting and blinding use two $\log(n)$-bit randomization factors. As pointed out by [106], the total amount of added noise entropy for shifting and blinding combined is $4\log(n)$ bits. For $n = 256$ this is equal to 32.

Algorithm 20 RLWE decryption using the shifting and blinding countermeasures.

```
Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function DEC($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $i, j, k, l \xleftarrow{\$} \{0, \ldots, n\}$
    $\mathbf{s} \leftarrow \omega^{-j}\phi^i\mathrm{NTT}(x^i) \odot \mathbf{s}$
    $\mathbf{c}_1 \leftarrow \omega^{-l}\phi^k\mathrm{NTT}(x^k) \odot \mathbf{c}_1$
    $\mathbf{c}_2 \leftarrow \omega^{j+l}\phi^{i+k}\mathrm{NTT}(x^{i+k}) \odot \mathbf{c}_2$
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$
    $\mathbf{d} \leftarrow \omega^{-(j+l)}\mathbf{d}$
    for $0 \le m < n$ do
        $d_{m+i+k} \leftarrow$ DECODE($d_m$)
```

6.4 New Protections

6.4.1 Shuffling

The point-wise multiplication $\mathbf{s} \odot \mathbf{c}_1$ in the NTT domain consists of n independent modular multiplications. The multiplications can be computed in any order. To compute

$$(s_0, s_1, \ldots, s_{n-1}) \odot (c_0, c_1, \ldots, c_{n-1}) = (s_0c_0, s_1c_1, \ldots, s_{n-1}c_{n-1})\,, \tag{6.4}$$

instead of computing s_ic_i for i starting from 0 to $n-1$, one could generate a random permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ and compute $s_{\sigma(i)}c_{\sigma(i)}$ for i from 0 to $n-1$. The computations do not change, but the moment at which each s_ic_i is computed, is randomized. This complicates the analysis of the obtained power traces. The operations that can be shuffled in this manner are not limited to the point-wise multiplication with $\mathbf{c}_1$ and subtraction of $\mathbf{c}_2$. The NTT consists of $\log_2(n)$ stages of $\frac{n}{2}$ butterfly operations each. The butterfly operations in a given stage are independent of each other, and can thus also be computed in any order. The same holds for the coefficient-wise decoding of $\mathbf{d} = \mathbf{c}_2 - \mathbf{s}\mathbf{c}_1$.

We consider two methods of shuffling the computations. The first of the two shuffling methods proposed in this section consists of replacing loop counters by linear feedback shift registers (LFSR). An LFSR is parametrized by an irreducible polynomial $f \in \mathbb{F}_2[x]$ and its degree k. It computes $x^i a \bmod f$ for $0 \leq i < 2^k - 1$ and some initial state $a \in \mathbb{F}_2[x]/f$. The computed values are exactly all the $2^k - 1$ invertible elements of the finite field $\mathbb{F}_2[x]/f$. The order in which they are computed is determined by the initial state a. Multiplication by x in $\mathbb{F}_2[x]/f$ is very fast and can be computed using only 1 shift and a XOR on bit positions depending on f. Our second shuffling method consists of generating a random permutation using a permutation network in the style of [18].

Algorithm 21 RLWE decryption using the shuffling countermeasure.

```
    Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
    Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    $\sigma \leftarrow$ RandomPermutation()
    $\mathbf{d} \leftarrow \sigma(\mathbf{c}_2) - \sigma(\mathbf{c}_1)\sigma(\mathbf{s})$
    for $0 \leq k < n$ do
        $d_{\sigma(k)} \leftarrow$ Decode($d_{\sigma(k)}$)
```

LFSR method

Let an LFSR be parametrized by some irreducible polynomial f of degree $\frac{n}{2}$. We let $k = \log_2(n) - 1$ and consider the coefficients vectors of polynomials in $\mathbb{F}_2[x]/f$ to be the binary representations of integers ranging from 0 to $\frac{n}{2} - 1$. The LFSR thus generates a sequence of $\frac{n}{2} - 1$ integers that will serve as indices for the loop counter in Algorithm 13. Instead of computing the i-th butterfly operation at the i-th loop iteration, we compute the butterfly operation that is on the j-th position, where j is the index corresponding to the i-th element generated by the LFSR. In other words, the normal loop counter is replaced by an LFSR. The LFSR has only $2^k - 1$ outputs, whereas we need 2^k for the $\frac{n}{2}$ butterfly operations. Therefore the first operation of each stage is not shuffled: it is always computed in the first loop iteration of the stage.

To obtain a meaningful permutation, we use the PRNG to generate a new initial state a at the start of each stage. Since $a = 0$ is not allowed as initial state, we set $a = 1$ as the default state in the case that the PRNG outputs 0. The initial state is thus set to default state with probability $\frac{4}{n}$. All the other initial states appear with probability $\frac{2}{n}$. This slight bias could be reduced by having the PRNG generate multiple initial states and selecting a non-zero state.

The $2^k - 1$ possible initial states determine $2^k - 1$ unique sequences. The operations of a complete $\log_2(n)$ stage NTT can then be computed in $(2^k - 1)^{\log_2(n)}$ different ways. For $n = 256$ and $k = 7$ this is more than 2^{55}. With the LFSR method applied to the pointwise operations outside the NTT as well, the total number of random bits added is equal to 71. A single trace attack like [99] that requires all of the $\log_2(n)$ stages seems unlikely to succeed on an implementation using the LFSR countermeasure as described.

We use an LFSR of degree $k = \log_2(n)$ in a similar manner to shuffle the order of the n pointwise multiplications outside the NTT.

Drawbacks to the LFSR loop counter include a limited permutation space, a slightly biased outcome and the fact that the first element is not permuted.

Permutation Network Method

We propose to use a permutation network generator in the style of [18]. Their permutation generator is designed for use in AES and is impractical for larger ($N = 256$) permutations. It is also biased. We simplify their permutation network to obtain a permutation generator that can generate $N^{N/2}$ permutations and that is uniform on its range. In the remainder of this section, the parameter N is the size of the permutation, which is equal to n for the shuffling of the pointwise operations. To shuffle the butterfly operations during the computation of the NTT, N is substituted by $\frac{n}{2}$.

For $b \in \{0, 1\}$, let the operators T_b be defined by the mapping:

$$\begin{aligned} T_b : \{0, \ldots, N-1\} &\longrightarrow \{0, \ldots, N-1\} \\ x &\longmapsto \left\lfloor \frac{x}{2} \right\rfloor + b\frac{N}{2} \end{aligned}$$

Then T_0 is a bitwise shift erasing the least significant bit (LSB), and T_1 applies the same shift and sets the MSB to 1.

The permutation network consists of $k = \log_2(N)$ stages and takes as input $(x_0, \ldots, x_{N-1}) = (0, \ldots, N-1)$. During each stage, $\frac{N}{2}$ random bits $b_1, \ldots, b_{N/2}$ are generated and for all pairs (x_{2i}, x_{2i+1}) the images $T_{b_i}(x_{2i})$ and $T_{\bar{b}_i}(x_{2i+1})$ are computed. In other words, for each pair (x_{2i}, x_{2i+1}), one is sent to position i, while the other is mapped to $i + \frac{N}{2}$. This is equivalent to writing one bit of the image of x_{2i} under the generated permutation and writing its complement to the image of x_{2i+1}. The network is shown for $N = 8$ in Figure 6.8. It is exactly the same as the computation scheme of the constant geometry NTT, in which the butterfly operators are replaced by controlled swap operators.

For any integer $0 \leq x < N$ the image of x under the generated permutation can be written as $T_{b_1} \circ \cdots \circ T_{b_k}(x)$ and is equal to the value corresponding to the binary representation $(b_1, \ldots, b_k)_2$. The $\frac{kN}{2}$ control bits thus determine the image of each index under the generated permutation. By uniqueness of binary representation it follows that any modification to any subset of the $\frac{kN}{2}$ control bits

Figure 6.8: Permutation network for $N = 8$. Each box is an instance of the operator shown in Figure 6.9.

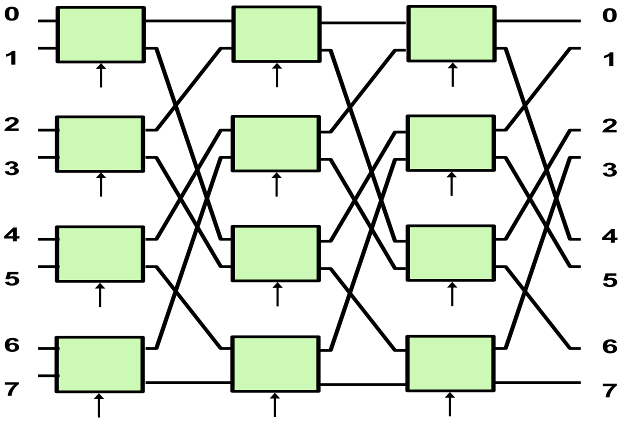
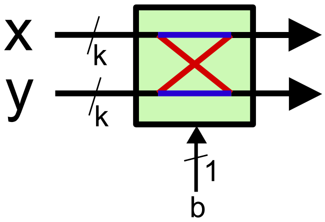


Figure 6.9: The swap operator is controlled by a random bit and swaps the inputs if this bit is equal to 1.



would modify the generated permutation as well. The permutation generator is therefore an injective map from $\{0,1\}^{Nk/2}$ into the set of all permutations Σ_N. This means that the number of possible configurations of the $\frac{kN}{2}$ control bits, which is equal to $N^{N/2}$, is exactly the number of permutations that can be generated by the network. The number of different permutations that can be obtained is 2^{1024} for the pointwise operations and 2^{256} for the NTT. Moreover, the output of the permutation network is uniform on its range given uniformly random input.

Since the permutation space is much larger than the one we obtain with the LFSR, we will only generate one $\{0,\ldots,\frac{n}{2}-1\} \to \{0,\ldots,\frac{n}{2}-1\}$ permutation for the NTT at the start of each decryption. Each stage is then computed in the order defined by this permutation. We also compute only one permutation of size n that will be used for all the pointwise operations during one decryption.

6.4.2 Randomization using Redundant Number Representation

In RSA and ECC, some exponent or scalar randomization countermeasures have been proposed against SCAs (see for instance [34]). A secret exponent or scalar can be randomized without loss of information by adding a random multiple of the group order to it. The corresponding power traces are thus randomized, removing correlation between the side channel traces and the secret key. A similar concept can be applied to RLWE.

We can add random multiples of the modulus q to the secret key coefficients without invalidating the secret key. This is done at the start of each decryption (see Algorithm 22). The PRNG is used to generate small r-bit random numbers for some integer parameter r. These numbers are multiplied by q and then added to the input and to the secret key. We then continue using arithmetic operations in $\mathbb{Z}/(2^r q)\mathbb{Z}$ instead of in $\mathbb{Z}_q$. The fact that for all $a, b \in \mathbb{Z}$ we have that $ab \bmod (2^r q) \equiv ab \bmod q$, ensures us that the result is in the correct equivalence class.

Algorithm 22 RLWE decryption using redundant secret key representation.

```
    Input: Secret key $\mathbf{s} \in \mathcal{R}_q$, ciphertext $(\mathbf{c}_1, \mathbf{c}_2) \in \mathcal{R}_q^2$
    Output: Plaintext $\mathbf{d} = (d_0, \ldots, d_{n-1}) \in \{0,1\}^n$
function Dec($\mathbf{c}_1, \mathbf{c}_2, \mathbf{s}$)
    for $0 \le k < n$ do
        $i, j \xleftarrow{\$} \{0, \ldots, 2^r - 1\}$
        $s_k \leftarrow s_k + i \cdot q$
        $c_k \leftarrow c_k + j \cdot q$
    $\mathbf{d} \leftarrow \mathbf{c}_2 - \mathbf{c}_1\mathbf{s}$                    ▷ compute in $\mathbb{Z}_{2^r q}$
    for $0 \le k < n$ do
        $d_k \leftarrow \text{Decode}_r(d_k)$
```

The redundancy is not removed for the decoding. Instead we modify the algorithm to decode the coefficients directly from $\mathbb{Z}/(2^r q)\mathbb{Z}$ to $\{0,1\}$. The new decoder (Figure 6.11) returns 0 if the input lies in the union of sets $\bigcup_{i=0}^{2^r} \left[\frac{-q}{4} + iq, \frac{q}{4} + iq\right)$ and returns 1 if the input is in $\bigcup_{i=0}^{2^r} \left[\frac{q}{4} + iq, \frac{3q}{4} + iq\right)$.

Figure 6.10: Architecture with our redundant representation countermeasure. Before the decryption, small r-bit random multiples of q are added to the coefficients of $\mathbf{c}_1$ and $\mathbf{s}$. The operations in the decryption function are performed in $\mathbb{Z}/(2^r q)\mathbb{Z}$.

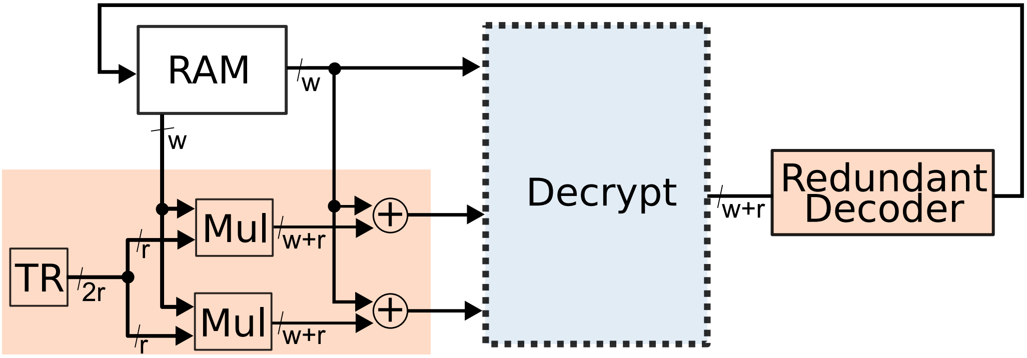
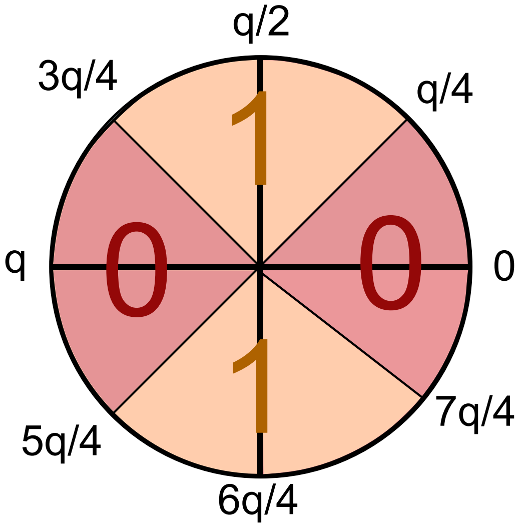


Figure 6.11: Specialized decoder for $r = 1$. To decode some $d \in \mathbb{Z}/2q\mathbb{Z}$, the ring is divided up in 8 intervals.



At each execution of the algorithm, the multipliers, adders and decoder are handling different inputs. The computations (and the corresponding traces) are thus randomized. A total of $256r$ random bits are added to the operands.

Validation through Correlation Power Analysis Simulations.

We evaluate the robustness of our countermeasure based on a redundant representation by simulating power attacks under assumptions favorable to the attacker. The polynomial multiplication in the NTT domain consists of n independent multiplications in $\mathbb{Z}/q\mathbb{Z}$. They are of the form $c \cdot s \bmod q$, where s is a coefficient of the secret key and c is a coefficient of the input ciphertext. We simulate correlation attacks on one modular multiplication of a known input coefficient c with an unknown secret key coefficient s.

We assume that the attacker observes the modular multiplication $c \cdot s \bmod q$ for a number of different (known) inputs c. For each modular multiplication she/he obtains the exact Hamming weight (HW) of the result. The attacker computes the "predictions": the HW of the value $c \cdot s \bmod q$ for all subkey candidates $s \in \mathbb{Z}_q$ and for all inputs c. She/he evaluates the correlation between the observed HW and the predictions. For each subkey possibility $\tilde{s} \in \mathbb{Z}_q$, the Pearson's correlation coefficient between the observed HW and the predictions is computed. Without countermeasures, the highest correlation is obtained for the correct subkey guess.

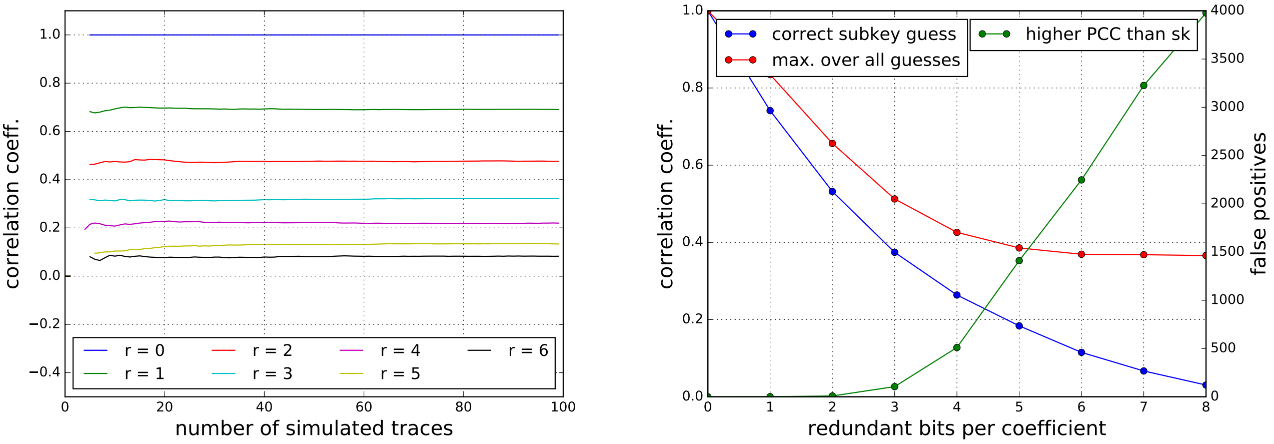
The inputs are randomized by adding a multiple of q and used in computations in $\mathbb{Z}/(2^r q)\mathbb{Z}$ for some redundancy parameter r. The impact of our countermeasure on the effectiveness of the power analysis can be seen (for $q = 7681$) in Figure 6.12. Without redundancy ($r = 0$), the attacker observes the exact HW of the value $c \cdot s \bmod q$ for different values of c. These HWs coincide with the predictions for the correct subkey guess, resulting in a correlation coefficient of 1. For higher levels of redundancy, the average of the correlation coefficient for the correct subkey guess decreases.

The right side of Figure 6.12 shows that the maximum correlation is obtained for incorrect subkey guesses for all $r \geq 1$. We refer to subkey guesses that yield to a higher correlation coefficient than the correct subkey guess as *false positives*. The number of these false positives increases with the redundancy level. For $r = 8$ and $r = 9$ there are on average around $\frac{q}{2}$ subkey guesses, for one coefficient of the secret key, that yield to a higher correlation coefficient than the correct key guess. Our countermeasure ensures that an exponential number of up to $\left(\frac{q}{2}\right)^n$ guesses have to be tested to recover the complete secret key.

6.5 Comparison of all Protections

FPGA implementation results for RLWE solutions with various countermeasures are presented in Table 6.1. Results from [102] are reported, and we also re-implemented their solution on an Artix-7 XC7A200 (denoted "A7") to provide fair comparisons. We also implemented the blinding and

Figure 6.12: Mean correlation over 1000 simulations between the correct subkey guess and the observed HW as a function of the number of traces (left) and the number of redundant bits per coefficient (right). Right: average (1000 simulations of 100 traces each) of the maximum correlation over all subkey guesses is shown in red and the number of subkey guesses with higher correlation than the correct secret key in green.



shifting methods from [106] and our shuffling methods. To the best of our knowledge, these are the first FPGA implementations for these countermeasures. Finally, the results for masking with our new masked decoder and our redundant randomized countermeasures are reported. The amount of randomness added for each countermeasure is specified in the second column of the table.

We cannot directly compare our re-implementation of the masking from [102] and their original results on a Virtex-II XC2VP7 (denoted “V2”). However, it can be seen that the impact of masking on the performance of their V2 implementation is very high compared to our A7 re-implementation. The computation time for decryption is tripled. This is probably because the number of arithmetic operations in $\mathbb{Z}_q$ is doubled while no parallelism is used. Moreover, it seems that their masked decoder is implemented sequentially. In our re-implementation of the masked decoder from [102], we use parallelism to significantly reduce the performance penalty of their 16-step decoder. This increases the area.

Our new masked decoder is relatively simple and requires a small area (about 20% reduction compared to the re-implementation of the decoder from [102]), with almost the performance of the unprotected implementation. Compared to the unprotected solution, we use extra DSP blocks and BRAMs to compute the decryptions of the two shares in parallel.

The blinding implementation gives a slightly slower solution. Its area overhead is smaller than for both masking techniques. However, we stress that this blinding countermeasure should be used in combination with another countermeasure (as specified in [91]), since the blinding factor is removed before the decoding step. The shifting implementation yields to similar overhead (although with lower

Table 6.1: FPGA results for RLWE with various countermeasures and $(q, n) = (7681, 256)$. The source column refers to the work in which the countermeasure was first proposed in LWE context. Timing and area results are for the decryption only.

Counter-measure	Entropy added (bits)	Src.	Impl.	FPGA	Lat.	Clk. (ns)	Time (μ s)	Slice, LUT, DSP, BRAM
None	0	-	[102]	V2	2800	8.3	23.5	-, 1713, 1, -
Masking	3328	[102]	[102]		7500	10	75.2	-, 2014, 1, -
None	0	-	this work	A7	2357	3.3	7.8	483, 1163, 2, 3
Blinding	16	[106]			2768	3.8	10.6	941, 2284, 3, 4
Shifting	16	[106]			3138	4.7	14.8	832, 2150, 3, 4
Shift + Blind	32	[106]			3183	4.6	14.7	1063, 2781, 3, 4
Masking	3328	[102]			2517	4.0	10.1	2187, 5500, 5, 6
Our Mask.	3328	this work			2510	4.0	10.1	1722, 4269, 5, 6
Permutation	1280				2521	4.5	11.4	3183, 7385, 2, 4
LFSR ctr.	71				2846	3.6	10.3	1069, 2861, 2, 3
$r = 1$	256				2272	3.7	8.5	629, 1599, 2, 3
$r = 2$	512				2273	3.6	8.2	611, 1664, 2, 3
$r = 3$	768				2333	3.8	8.9	807, 2067, 2, 3
$r = 4$	1024				2338	3.6	8.5	872, 2285, 2, 3
$r = 5$	1280				2352	3.8	9.0	990, 2677, 2, 6
$r = 6$	1536				2394	3.9	9.4	1254, 3466, 3, 6
$r = 7$	1792				2410	3.9	9.4	1713, 5017, 3, 6
$r = 8$	2048				2426	3.9	9.5	2544, 7837, 3, 6

frequency) and its combination with blinding seems to be worthwhile. The permutation network is relatively costly in area. The LFSR loop counter is cheaper and slightly faster.

Finally, our redundant randomized countermeasure does not need additional DSPs or BRAMs to be implemented for small redundancy parameters ($r \leq 4$) and can therefore be used as a cheap way to secure the decryption. For higher redundancy levels the multiplication cannot be computed within a single 18×25 bits multiplier, as the ones hardwired in the Artix DSP blocks. A few additional DSP blocks and BRAMs are needed.

6.6 Generalizing the Countermeasures to Apply to MLWE/LWE

In this section we describe how to apply similar protections to MLWE and LWE based implementations. This work was not part of our IndoCrypto 2019 publication [117].

6.6.1 Masking

MLWE In the MLWE based decryption algorithm the secret key consists of a vector $\mathbf{s} \in \mathcal{R}_q^k$ for $k > 1$. The secret key $\mathbf{s}$ is split up into two parts $\mathbf{s} = \mathbf{s}' + \mathbf{s}''$ for some uniformly random $\mathbf{s}' \xleftarrow{\$} \mathcal{R}_q^k$. For the vector multiplication between the input ciphertext $\mathbf{c}_1$ (which is a row vector) and the secret key (column vector), we then have:

$$\begin{aligned}\mathbf{c}_1\mathbf{s} &= \mathbf{c}_1(\mathbf{s}' + \mathbf{s}'') \\ &= \mathbf{c}_1\mathbf{s}' + \mathbf{c}_1\mathbf{s}''\end{aligned}$$

Vector multiplication is distributive with respect to addition, so no particular modification to the masking scheme is needed. The only difference with the RLWE case is that we now need to sample a uniform random $\mathbf{s}'$ from $\mathcal{R}_q^k$ instead of $\mathcal{R}_q$. The masked decoding algorithm for RLWE described in this chapter, can also be used for the MLWE case.

LWE The distributivity with respect to addition also holds for the LWE case, where the operands are matrices over $\mathbb{Z}_q$. The decoding however, is slightly different. The parameters for LWE for which we implemented the cryptosystem in the previous chapter, include the encoding parameter B. This parameter determines how many plaintext bits are encoded in each coefficient of the ciphertext part $\mathbf{c}_2$. In RLWE and MLWE, we have $B = 0$ such that each coefficient encodes $B + 1 = 1$ plaintext bit. In LWE, the plaintext coefficients are $(B + 1)$-bit postive integers that are, once encoded, added to the ciphertext part $\mathbf{c}_2$. For $B = 1$, the plaintext coefficients are in $\{0, 1, 2, 3\}$ and are encoded to $0, \left\lfloor \frac{q}{4} \right\rfloor, \left\lfloor \frac{q}{2} \right\rfloor$ and $\left\lfloor \frac{3q}{4} \right\rfloor$ respectively. To decode some $d \in \mathbb{Z}_q$, it has to be determined in which of the four intervals $\left\{ \left\lfloor \frac{i \cdot q}{4} \right\rfloor - \left\lfloor \frac{q}{8} \right\rfloor, \ldots, \left\lfloor \frac{i \cdot q}{4} \right\rfloor + \left\lfloor \frac{q}{8} \right\rfloor \right\}$ for $0 \leq i < 4$ it lies. The masked decoder therefore defines 8 regions $R_0, \ldots, R_7$, where $R_i = \left\{ \left\lfloor \frac{iq}{8} \right\rfloor, \ldots, \left\lfloor \frac{(i+1)q}{8} \right\rfloor - 1 \right\}$ for $0 \leq i < 8$. If $d' \in R_i$ and $d'' \in R_j$ then:

$$\left\lfloor \frac{iq}{8} \right\rfloor \leq d' < \left\lfloor \frac{(i+1)q}{8} \right\rfloor \tag{6.5}$$

and

$$\left\lfloor \frac{jq}{8} \right\rfloor \leq d'' < \left\lfloor \frac{(j+1)q}{8} \right\rfloor, \tag{6.6}$$

from which it follows that:

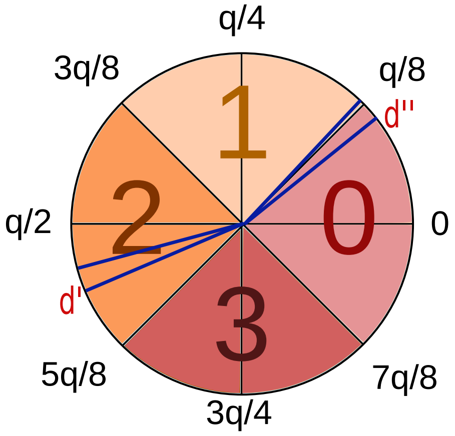
$$\left\lfloor \frac{iq}{8} \right\rfloor + \left\lfloor \frac{jq}{8} \right\rfloor \leq d' + d'' < \left\lfloor \frac{(i+1)q}{8} \right\rfloor + \left\lfloor \frac{(j+1)q}{8} \right\rfloor. \tag{6.7}$$

Then we have

$$\left\lfloor \frac{(i+j)q}{8} \right\rfloor - 1 < d' + d'' < \left\lfloor \frac{(i+j+2)q}{8} \right\rfloor. \tag{6.8}$$

This means that, similar to the case where $B = 0$, we have a useful property:

Figure 6.13: Refreshing the shares when $B = 1$ can be done by computing the distances to the interval bounds.



Property 6.6.1. *If $d' \in R_i$ and $d'' \in R_j$ then $(d' + d'') \in R_{i+j \bmod 8} \cup R_{i+j+1 \bmod 8}$.*

The unions of sets $R_{2i-1 \bmod 8} \cup R_{2i}$ contains all the pre-images of i under $\textsc{Encode}_1(\cdot)$. Let $d' \in R_i$ and $d'' \in R_j$ for some $i, j < 8$. If $i + j \equiv 1 \bmod 2$, then the property allows to decode the two shares without computing their sum. Their sum lies in $R_{i+j \bmod 8} \cup R_{i+j+1 \bmod 8}$, which can be written as $R_{2l-1 \bmod 8} \cup R_{2l}$ for some integer l.

The hard cases are given by the $(d', d'') \in R_i \times R_j$ for which $i + j \equiv 0 \bmod 2$. They can be reduced to easy cases by refreshing the shares using the methods described in sections 6.2 and 6.3.1. The resharing method from [102], which adds and subtracts some random constants from the two shares, can be applied without modification. The specific values presented by [102] however, are optimized for the case where $B = 0$, and they might be less effective for $B > 0$. The decryption failure rate would then increase. To avoid this, one could adapt our deterministic masked decoding algorithm in the following manner. Let $(d', d'') \in R_i \times R_j$ be a hard case and compute the distances with the region bounds: $\delta' = \left\lfloor \frac{(i+1)q}{8} \right\rfloor - d'$ and $\delta'' = d'' - \left\lfloor \frac{jq}{8} \right\rfloor$. If $\delta' < \delta''$, then d' can be shifted to an adjacent region by adding the resharing constant δ'. The other share d'' is still in the same region after subtracting δ', because

$$\left\lfloor \frac{jq}{8} \right\rfloor = d'' - \delta'' < d'' - \delta' < d'' < \left\lfloor \frac{(j+1)q}{8} \right\rfloor . \tag{6.9}$$

The refreshed shares then lie in R_{i+1} and R_j respectively. The property 6.6.1 can be used to verify that this is indeed an easy case. The same precautions from section 6.3.1 with respect to comparison operators have to be applied. Figure 6.13 shows an example of a hard case with shares in R_0 and R_4. According to property 6.6.1 the sum of the shares lies in $R_4 \cup R_5$, which can decode to either 2 or 3. Shifting one share to the adjacent region and re-applying property 6.6.1 shows that the sum of the shares lies in $R_5 \cup R_6$, and therefore decodes to 3.

The general case $B > 1$ is treated in the same manner. The ring $\mathbb{Z}_q$ is divided up into 2^{B+2} regions $R_i = \left\{ \left\lfloor \frac{iq}{2^{B+2}} \right\rfloor, \dots, \left\lfloor \frac{(i+1)q}{2^{B+2}} \right\rfloor - 1 \right\}$ for $0 \leq i < 2^{B+2}$. The indices i and j of regions R_i and R_j for which $d' \in R_i$ and $d'' \in R_j$ are determined. The analysis from equations (6.5), (6.6), (6.7) and (6.8), can be generalized by simply changing the denominator, the equations remain valid. A generalization of property 6.6.1 can thus be formulated:

Property 6.6.2. *If $d' \in R_i$ and $d'' \in R_j$ then $(d' + d'') \in R_{i+j \bmod 2^{B+2}} \cup R_{i+j+1 \bmod 2^{B+2}}$.*

If $i + j \equiv 1 \bmod 2$, then the shares form an easy case that can be resolved by applying the generalized property. The hard cases can be solved by shifting one of the shares to an adjacent region while keeping the other in the same region.

6.6.2 Blinding

MLWE The blinding countermeasure can be implemented in exactly the same way as for RLWE. The blinding scalars a and b are multiplied by vectors $\mathbf{s} = \left(\mathbf{s}^{(i)}\right)_{1 \leq i \leq k}$ and $\mathbf{c}_1\left(\mathbf{c}^{(i)}\right)_{1 \leq i \leq k}$. The vector multiplication computes:

$$\sum_{i=1}^{k} b\mathbf{c}^{(i)} \cdot a\mathbf{s}^{(i)} = ab \sum_{i=1}^{k} \mathbf{c}^{(i)} \cdot \mathbf{s}^{(i)}. \tag{6.10}$$

It follows that the blinding can be undone by multiplying the result by the inverse of ab. The pre-computed powers of the roots of unity can be used as blinding factors to avoid the modular inversion. Note that for a given vector, each polynomial is multiplied by the same blinding constant. It is not possible to use different blinding factors for each polynomial, since equation (6.10) would not hold and it would therefore be impossible do undo the blinding.

LWE Scalar multiplication with the secret key matrix and the ciphertext input matrix have similar behaviour as in $\mathcal{R}_q$. The blinding can thus be implemented in the same way as for RLWE. For LWE however, there are no pre-computed powers of the roots of unity. Moreover, since q is a power of 2, not all elements are invertible. One would need to implement an algorithm for modular inversion of the invertible elements in $\mathbb{Z}_q$. Another option would be to pre-compute a list of the form $\{g, g^2, g^3, \dots, g^{\mathrm{ord(g)}}\}$ for some $g \in \mathbb{Z}_q$ of high order.

6.6.3 Shifting

MLWE Using the associativity of scalar multipication in $\mathcal{R}_q^k$, the shifting countermeasure can be applied to MLWE by multiplying $\mathbf{s}$ and $\mathbf{c}_1$ by random powers a, b of x:

$$\sum_{i=1}^{k} x^a\mathbf{c}^{(i)} \cdot x^b\mathbf{s}^{(i)} = x^{a+b} \sum_{i=1}^{k} \mathbf{c}^{(i)} \cdot \mathbf{s}^{(i)}. \tag{6.11}$$

Algorithms 19 and 20 can be generalized to the MLWE case by applying the shift (in the NTT domain) to each polynomial seperately.

LWE The structure of the polynomial ring $\mathcal{R}_q$ is required to shift the coefficients in the coefficient vectors. Due to the absence of such a structure in the case of LWE, this countermeasure cannot be applied.

6.6.4 Shuffling

MLWE Shuffling can be applied on multiple levels in the MLWE decryption algorithm. Permutation of the order of the computation of the point-wise multiplications and the butterfly operations, can be used without any modification as described in section 6.4.1. Additionally, it is possible to shuffle the operations on a vector level. That is, the multiplication

$$\begin{pmatrix} \mathbf{c}^{(1)} & \dots & \mathbf{c}^{(k)} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{s}^{(1)} \\ \vdots \\ \mathbf{s}^{(k)} \end{pmatrix} = \mathbf{c}^{(1)}\mathbf{s}^{(1)} + \dots + \mathbf{c}^{(k)}\mathbf{s}^{(k)} \tag{6.12}$$

can be shuffled by computing the partial products $\mathbf{c}^{(i)}\mathbf{s}^{(i)}$ in a random order. To do this, a random permutation $\sigma : \{1, \dots, k\} \to \{1, \dots, k\}$ has to be generated. Since k is very small ($k = 2, 3$ or 4), this can be done at minimal cost.

LWE The matrix multiplication of the input ciphertext matrix with the secret key matrix can be shuffled in multiple ways. On a matrix level:

$$\begin{aligned}
\mathbf{CS} &= \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0(k-1)} \\ c_{10} & c_{11} & \cdots & c_{1(k-1)} \\ \vdots & \vdots & \vdots & \vdots \\ c_{(m-1)0} & c_{(m-1)1} & \cdots & c_{(m-1)(k-1)} \end{pmatrix} \times \begin{pmatrix} s_{00} & \cdots & s_{0(m-1)} \\ s_{10} & \cdots & s_{0(m-1)} \\ s_{20} & \cdots & s_{0(m-1)} \\ \vdots & \vdots & \vdots \\ s_{(k-1)0} & \cdots & s_{0(m-1)} \end{pmatrix} \\
&= \mathbf{C} \times \begin{pmatrix} s_{00} \\ s_{10} \\ \vdots \\ s_{(k-1)0} \end{pmatrix} + \mathbf{C} \times \begin{pmatrix} s_{01} \\ s_{11} \\ \vdots \\ s_{(k-1)1} \end{pmatrix} + \dots
\end{aligned}$$

The m partial matrix-vector products can be computed in any order. A permutation σ of size m can be used to randomize this order.

Each of these partial matrix-vector product consists of m partial vector-vector multiplications:

$$\mathbf{C} \times \begin{pmatrix} s_{0i} \\ s_{1i} \\ \vdots \\ s_{(k-1)i} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} & \dots & c_{0(k-1)} \end{pmatrix} \times \begin{pmatrix} s_{0i} \\ s_{1i} \\ \vdots \\ s_{(k-1)i} \end{pmatrix} + \dots$$

Another random permutation τ of size m allows to randomize the order of these m operations as well. And each vector multiplication consists of k modular multiplications:

$$\begin{pmatrix} c_{j0} & c_{j1} & \dots & c_{j(k-1)} \end{pmatrix} \times \begin{pmatrix} s_{0i} \\ s_{1i} \\ \vdots \\ s_{(k-1)i} \end{pmatrix} = c_{j0}s_{0i} + c_{j1}s_{1i} + c_{j2}s_{2i} + \dots$$

A third permutation ρ of size k may be generated to randomize these modular multiplications.

6.6.5 Redundant number representation

The countermeasure adds a random multiple of the modulus q to each coefficient of the secret key and the input ciphertext. This countermeasure operates on the coefficient level only and is independent of the ring/module structure, so that it can be applied directly to both LWE and MLWE decryption.

6.7 Conclusion and Discussion

In this chapter, we compared several countermeasures against SCAs for RLWE from [102], [106] and proposed new ones. Our first proposed countermeasure is an adaptation of [102] with a new masked decoder which is deterministic. Our second one uses a redundant representation to randomize polynomial coefficients. We also implemented two different methods for shuffling. All the countermeasures (from literature and our ones) have been implemented on FPGA to evaluate the overhead compared to a common reference implementation on the same FPGA. Our new decoder uses over 20% less slices and LUTs than the one from [102]. To the best of our knowledge, we also present the first FPGA implementations for the blinding and shifting countermeasures from [106], and a combination of the two. Finally, our protection based on redundancy at ring level provides a cheap randomization method with an adjustable security/overhead trade-off. We described the modifications to be applied to the countermeasures in order to apply to LWE and MLWE cryptosystems.

Chapter 7

Conclusion

Post-quantum cryptography is becoming increasingly important as the development of quantum computers continues to advance. Its deployment must take place well before the existence of quantum computers that are sufficiently large to break RSA and ECC instances. This has led to the NIST post-quantum contest, an effort to select and standardize public-key encryption and digital signature algorithms that can resist against quantum attacks. Candidate cryptosystems based on lattice problems, in particular LWE and its variants over structured lattices, are among the most promising contenders. The standardized algorithms will need to be implemented in constraint environments or with strict timing and cost requirements. Hardware acceleration using FPGAs may provide a practical solution. It is therefore imperative to evaluate the implementation cost of the post-quantum algorithms, and to analyse how the implementations can be protected against side-channel attacks.

In this thesis, we aimed to provide a fair and objective comparison between different cryptographic algorithms contending in the NIST post-quantum contest. We used HLS tools to evaluate the performance and implementation cost of the algorithms. We improved the performance of HLS generated implementations for finite field arithmetic, crucial in PKC. This was achieved by replacing the basic modular reduction operator in the C source code by customized algorithms, exploiting the fact that the modulus is constant. We also showed how to change the coding style of the C source code in order to obtain more performant implementations from HLS.

We implemented various lattice based cryptosystems on FPGA, and presented the first FPGA implementation of MLWE based public-key encryption. We compared the performances of LWE, RLWE and MLWE based algorithms. Our implementations have in many cases better computation time/area trade-offs than the ones from the state of the art. Our CCA secure MLWE implementation for instance, uses than 28 times fewer DSPs than the MLWR implementation from [44] while computing the encryption algorithm faster. This may have implications in the NIST post-quantum contest, where computation time and area utilization are important factors besides theoretical security. The MLWE based KEM Kyber seems to be more efficient in hardware than the MLWR based Saber. We suspect that the major cause of this difference is the fast polynomial multiplication using the NTT, which can be used in MLWE based algorithms but not in MLWR. We also evaluated the efficiency of the use of parallelism in the implementation of LWE, RLWE and MLWE. We found that LWE, while easily parallelizable even for higher degrees of parallelism, still can not compete with the performances of the NTT based algorithms in RLWE and MLWE. It will be difficult to see a practical application of LWE in constraint environments. While its theoretical security inspires much confidence, its computation time is significantly higher than its counterparts computing over structured lattices. Another interesting result from our hardware exploration is the difference in throughput between our parallelized implementations of RLWE and MLWE. The structure of small vectors and matrices in MLWE facilitates the parallelization of operations when implementing on FPGA. Polynomial arithmetic can be computed independently for each of the vector indices, which means that these computations can be performed concurrently. For a vector containing k polynomials, the NTT of all polynomials can be computed simultaneously on k different DSP blocks. The computation time of such a parallelized MLWE implementation then approaches the computation time of RLWE for much smaller ($n = 256$ instead of 1024) parameters. The absence of such a structure in RLWE limits its potential for parallelization. MLWE based algorithms, such as Kyber, thus have an advantage over RLWE based algorithms such as NewHope.

We improved the robustness of our implementation of RLWE against side-channel attacks. To this end, we implemented SCA countermeasures from the state of the art, such as masking, blinding and shifting, and proposed various improvements, as well as new countermeasures at arithmetic level using redundant number representation. Relatively cheap countermeasures, such as shifting and blinding can be implemented by adding just one DSP block to compute the additional multiplications. We proposed a combination of shifting and blinding, and showed that this increases the security at limited cost in terms of area. The masking scheme from [102] made use of a non-deterministic decoder which resulted in a higher computation time and a higher probability of decryption failure. We proposed a deterministic decoder and implemented our solution on FPGA. Compared to a re-implementation of the original masking scheme, the proposed algorithm is implemented on a reduced area. We also proposed new countermeasures. The independence between certain operations in the decryption algorithm, allows to randomize the computation order of these operations. We proposed two different methods to randomize the computation order. The first one uses an LFSR in order to generate random permutations in a cheap manner. Our second proposition uses a permutation network to sample random permutations from a much larger subset of Σ_n. This is a more secure countermeasure, as it introduces more entropy to the randomization. We found that the permutation network countermeasure has a higher resource utilization when implemented on FPGA, compared to our LFSR countermeasure. The computation time however, is only 10 percent higher, resulting in an interesting trade-off between implementation cost and security againt side-channel attacks. We found similar trade-offs for our new countermeasure using redundant number representations. We proposed to represent coefficients in $\mathbb{Z}_q$ by randomly picked elements in the same equivalence class. By randomizing the representation of the secret key, side-channel attacks can be thwarted. Adding a higher degree of redundancy increases the security, but comes at a higher cost. Arithmetic for larger integers has to be implemented, thereby increasing the utilization of LUTs, DSPs and BRAMs.

Résumé en français

Introduction

La cryptographie permet, entre autres, l'échange sécurisé d'information. Supposons que deux personnes, Alice et Bob, souhaitent s'échanger une information confidentielle. L'échange se fait à distance, donc le contenu est susceptible d'être intercepté par une personne tierce, Ève, qui n'est pas autorisée à avoir accès au contenu. L'échange doit donc se faire d'une façon sécurisée, le contenu doit être protégé. Le *chiffrement* est un outil cryptographique qui permet de cacher le contenu (le *message*) dans un *texte chiffré*. Il nécessite une *clé*, permettant de chiffrer et déchiffrer des messages. Un texte chiffré ne peut être déchiffré qu'avec cette clé seule. Uniquement les personnes disposant de cette clé peuvent donc lire le contenu du message chiffré. De telles constructions relèvent de la *cryptographie symétrique*. Des algorithmes tels que AES et DES peuvent être utilisés pour réaliser des échanges confidentiels, à la condition qu'Alice et Bob disposent de la même clé. Le partage sécurisé d'une clé, un *échange de clés*, fait l'objet de la *cryptographie asymétrique*. La difficulté d'un échange de clés vient du fait que le contenu de cet échange risque également d'être intercepté par des personnes tierces. Le *chiffrement à clé publique* se sert de fonctions mathématiques à sens unique à trappe afin de surmonter cette difficulté. Un schéma de chiffrement à clé publique consiste en trois fonctions: la génération de clés $\mathcal{G}$, le chiffrement $\mathcal{E}$ et le déchiffrement $\mathcal{D}$. La fonction $\mathcal{G}$ retourne un couple (k, K) d'une clé secrète k et une clé publique K. Les fonctions doivent satisfaire la propriété de correction, c'est-à-dire, pour tout message m:

$$\mathcal{D}(k, \mathcal{E}(K, m)) = m.$$

Autrement dit, le déchiffrement d'un texte chiffré est égal au message en clair. En outre, étant donnée seulement la clé publique K et un texte chiffré $c = \mathcal{E}(K, m)$, il doit être pratiquement impossible de calculer m. Seule la clé secrète permet de retrouver le message clair. Le premier exemple d'un schéma de chiffrement à clé publique est le protocole RSA [104]. Dans RSA, la fonction $\mathcal{G}$ génère deux grands nombres premiers p et q et un nombre K aléatoire et inversible modulo $(p-1)(q-1)$, et retourne la clé publique K et la clé secrète $k = K^{-1} \bmod (p-1)(q-1)$. Le produit des nombres premiers $N = pq$ fait également partie de la clé publique. La fonction $\mathcal{E}$ prend en entrée un message en clair m et calcule $\mathcal{E}(m, K) = m^K \bmod N$. Pour déchiffrer un texte chiffré $c = m^K \bmod N$, on calcule $\mathcal{D}(c, k) = c^k \bmod N = m$. Un attaquant qui ne dispose pas de la clé secrète, n'a pas d'autre choix que de la calculer à partir de la clé publique, afin de déchiffrer un message. Pour cela, il a besoin de p et q, qu'il pourrait obtenir en factorisant le paramètre public $N = pq$. Le problème mathématique qui consiste à factoriser des grands nombres composés de nombres premiers, ne peut pas être résolu en temps polynomial par des algorithmes classiques. Pour des p et q suffisamment grands, il est donc quasiment impossible de trouver la clé secrète à partir de la clé publique. De même, il est impossible en pratique de trouver le message clair à partir du message chiffré sans connaître la clé secrète. Le cryptosystème RSA permet donc de réaliser des échanges de clé sécurisés. Dans le cas de la cryptographie sur des courbes elliptiques (ECC) c'est le problème du logarithme discret sur les courbes elliptiques qui garantit la sécurité théorique du cryptosystème.

Néanmoins, l'*algorithme quantique* de Shor [108] permet de résoudre le problème de factorisation en temps polynomial, en utilisant un *ordinateur quantique*. En conséquence, RSA peut être cassé par des attaquants disposant d'un ordinateur quantique. Il se trouve que d'autres protocoles d'échange de clé, tels que Diffie-Helmann (sur certains groupes), peuvent également être attaqué en utilisant l'algorithme de Shor. L'arrivée éventuelle d'un ordinateur quantique suffisamment performant, représenterait donc une menace pour la confidentialité des échanges numériques. La solution consiste à développer des protocoles d'échange de clé robustes face aux attaques quantiques. C'est pour cette raison qu'en 2016 le NIST a initié une compétition [36] pour des algorithmes cryptographiques *post-quantiques*, qui a pour objectif de sélectionner les meilleures solutions. La sécurité de ces nouveaux protocoles doit reposer sur la difficulté de problèmes mathématiques qui ne peuvent pas être résolus facilement par des algorithmes quantiques. Parmi les exemples, on peut citer des problèmes sur les *réseaux euclidéens*, tel que LWE [100]. Ce problème consiste à trouver un point du réseau $\mathbf{s}$, étant donné une base aléatoire $\mathbf{A}$ du réseau et une approximation du point $\mathbf{As}$ (voir la figure 7.1).

Les cryptosystèmes basés sur LWE, tel que FrodoKEM [28], ont un temps de calcul très élevé, car le chiffrement utilise une multiplication matricielle de grande dimension. Afin de reduire le temps de calcul, les variantes RLWE et MLWE du problème LWE ont été introduites par [81] et [75] respectivement. Dans ces variantes, les réseaux sont munis d'une structure algébrique. Cette structure algébrique permet de remplacer la multiplication matricielle par une multiplication polynomiale, qui peut être calculée à l'aide d'algorithmes performants, tel que la NTT. NewHope [4], basé sur RLWE, et Kyber [27], basé sur MLWE, sont deux cryptosystèmes présents dans la compétition du NIST.

En pratique, les applications qui requièrent l'utilisation d'algorithmes cryptographiques, ont des contraintes en termes de temps de calcul. L'accélération matérielle consiste à implanter en matériel des opérateurs dédiés, dans le but de reduire leur temps de calcul. La sécurité des implantations dépend aussi de leur vulnérabilité face aux *attaques par canaux auxiliaires* [71]. Ce type attaque utilise de l'information telle que le temps de calcul ou encore la consommation d'énergie, pour en déduire les valeurs des secrets manipulés dans l'algorithme implanté. La clé secrète, utilisée pendant le calcul

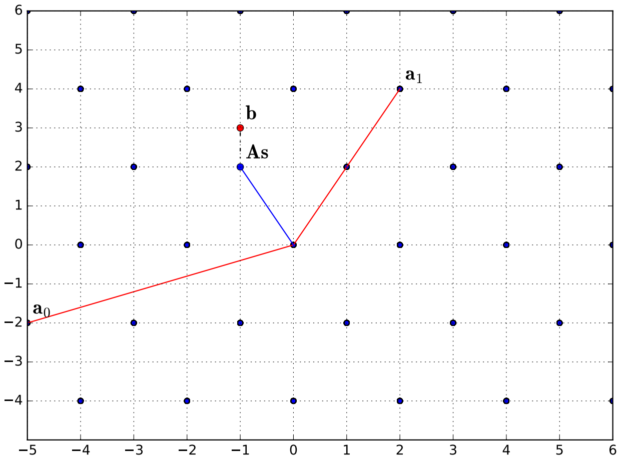


Figure 7.1: Le problème LWE sur un réseau éuclidéen d'une dimension 2 sur $\mathbb{Z}_{12}$: étant donnée une base aléatoire $\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1)$ et un point $\mathbf{b}$ proche du réseau, trouver $\mathbf{s}$ tel que $\mathbf{As}$ soit proche de $\mathbf{b}$.

de la fonction de déchiffrement, peut ainsi être découverte par un attaquant ayant accès au dispositif matériel.

Dans cette thèse, nous nous intéressons à l'accélération matérielle de schémas de chiffrement à clé publique basés sur des réseaux euclidéens. Nous implantons sur FPGA des cryptosystèmes basés sur LWE, RLWE et MLWE afin de comparer leurs performances. Nous étudions la sécurité matérielle des implantations et proposons des protections contre les attaques par canaux auxiliaires. Nous analysons les coûts et les performances de toutes nos implantations, dans le but de trouver les meilleurs compromis entre le niveau de sécurité, la performance et la surface.

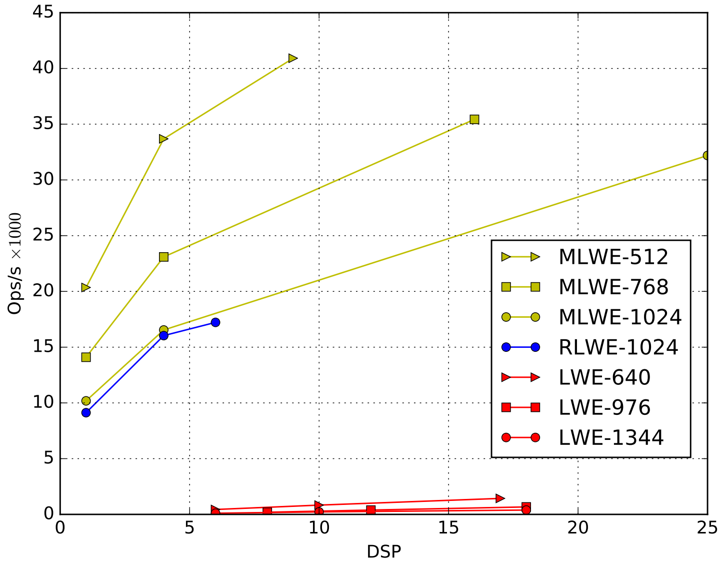
Contributions

Durant cette thèse, nous avons utilisé l'outil de synthèse de haut niveau Vivado HLS de Xilinx pour implanter des algorithmes cryptographiques sur FPGA. Ces algorithmes nécessitent des calculs arithmétiques dans des corps finis du type $\mathbb{F}_p$ pour un nombre premier p. Le calcul efficace de la réduction modulaire n'est que minimalement supporté par l'outil de synthèse. L'opérateur de réduction modulaire, représenté par le symbole % en language C, n'exploite pas le fait qu'un modulo soit fixe, ou qu'il ait une structure particulière, tels que les moduli avec une décomposition binaire très creuse. Nous avons donc amélioré la performance de l'implantation d'arithmétique modulaire en tirant profit de ces propriétés. Ces travaux ont fait l'objet d'une publication à la conférence francophone COMPAS [48]. Dans cet article, nous avons démontré que le temps de calcul et la surface des implantations de l'arithmétique modulaire peuvent être réduits en utilisant des algorithmes tels que Barrett [17], Montgomery [88] ou encore des algorithmes spécifiques dans le style de Solinas [109] pour des moduli creux. Nous avons aussi étudié l'impact du niveau de parallélisme, des *directives* et du style de code sur la performance des implantations obtenues avec l'outil HLS.

La deuxième contribution de cette thèse est l'optimisation et la comparaison d'implantations sur FPGA de cryptosystèmes basés sur LWE, RLWE et MLWE, sécurisés contre les attaques à clair connu (CPA) et les attaques à chiffré choisi (CCA). Nos implantations ont, dans la plupart des cas, des meilleurs compromis entre le temps de calcul et la surface que celles de l'état de l'art. Nous avons présenté la première implantation sur FPGA du cryptosystème basé sur MLWE. Cette implantation utilise 28 fois moins de DSP blocs que l'implantation de MLWR par [44], tout en ayant un temps de calcul plus faible. Il semble donc que la NTT, utilisée dans MLWE mais non pas dans MLWR, soit le meilleur algorithme pour le calcul arithmétique dans les réseaux euclidéens avec une structure algébrique favorable à l'utilisation de la NTT. Nous avons aussi étudié l'efficacité du parallélisme dans les implantations LWE, RLWE et MLWE (voir la figure 7.2). L'implantation LWE peut facilement être parallélisée, car l'opération principale du chiffrement LWE est une multiplication matricielle. Les multiplications de vecteurs qui constituent la multiplication matricielle, sont indépendantes les unes des autres, et peuvent donc être calculées en même temps. Cependant, même les implantations LWE qui utilisent beaucoup de parallélisme sont toujours moins rapide que les implantations RLWE/MLWE. Le chiffrement basé sur LWE ne convient donc pas pour les applications où il y a des contraintes strictes sur le temps de calcul. En comparant RLWE et MLWE, nous avons trouvé que la structure de MLWE permet un parallélisme plus efficace par rapport à RLWE. Les calculs dans le chiffrement basé sur MLWE se font sur des vecteurs de petite taille contenant des polynômes. Comme c'était le cas pour LWE, les opérations au niveau vecteur sont indépendantes les unes des autres, et peuvent donc être calculées simultanément. Au contraire, dans le chiffrement basé sur RLWE, le calcul le plus important en terme de complexité est la transformation NTT. Cette transformation, elle, est difficile à paralléliser à cause du schéma d'accès à la mémoire, qui est différent pour chacune des $\log_2(n)$ étapes de la NTT. Nous concluons donc, que MLWE est mieux adapté aux applications où le temps de calcul est prioritaire. Cette contribution fait l'objet d'une soumission au journal IEEE *Transactions on Computers*.

La troisième contribution a été présentée à la conférence internationale IndoCrypt en 2019 [117]. Dans ce travail, nous avons amélioré la sécurité de nos implantations face aux attaques par canaux auxiliaires. Pour cela, nous avons implanté plusieurs contremesures de l'état de l'art, telles que le masquage [102] et le *blinding* et *shifting* [106]. Le masquage consiste à générer un polynôme aléatoire $\mathbf{s}'$ pour masquer la clé secrète $\mathbf{s}$. On obtient ainsi deux polynômes aléatoires $\mathbf{s}'$ et $\mathbf{s}'' = \mathbf{s} - \mathbf{s}'$ dont la somme est égale à la clé secrète. L'algorithme de déchiffrement est alors utilisé deux fois: une fois avec $\mathbf{s}'$ comme clé secrète et après avec $\mathbf{s}''$. La dernière étape de l'algorithme de masquage de [102] consiste en un décodage probabiliste qui combine les deux résultats afin d'obtenir le message en clair. Nous avons amélioré cette étape de l'algorithme en proposant un algorithme déterministe, qui est plus performant. L'implantation des deux solutions montre que notre amélioration utilise moins de surface. Nos implantations du blinding et du shifting sont moins coûteuses en termes de surface

Figure 7.2: Nombre de chiffrements par seconde en fonction du nombre de blocs DSP utilisés.



que le masquage. Nous avons proposé une contremesure qui combine le blinding et le shifting, dont l'implantation est aussi performante que celle de shifting, tout en apportant plus d'aléa et donc plus de sécurité. Le *shuffling* est une contremesure qui a été suggérée par [99] pour contrer une attaque SPA. Cette contremesure consiste à calculer les opérations dans un ordre aléatoire. Nous avons proposé deux façons de réaliser le shuffling dans le contexte du déchiffrement RLWE. La première utilise un registre à décalage à rétroaction linéaire (LFSR), qui remplace le compteur dans les boucles de multiplication de vecteurs et dans les boucles à l'intérieur de la NTT. Cette proposition nécessite relativement peu de bits aléatoires, et peut être implanté à bas coût, comme le montrent nos résultats d'implantation. La seconde façon d'implanter le shuffling utilise un nouvel algorithme que nous avons proposé, qui permet de générer des permutations aléatoires. La sortie de cet algorithme est uniforme sur un large sous-ensemble de l'ensemble des permutations. Cette contremesure utilise jusqu'à trois fois plus de slices et de LUTs que la solution LFSR. La dernière contremesure que nous avons proposée est basée sur une représentation redondante des élements du corps fini $\mathbb{Z}_q$. Chacun de ses élements peut être représenté par tous les membres de ses classes d'équivalence. En calculant avec l'arithmétique de l'anneau $\mathbb{Z}_{2^r q}$ pour un entier $r \geq 0$, chaque élement a 2^r représentations différentes. Nous randomisons les calculs en ajoutant des multiples aléatoires de q aux coefficients de la clé secrète, tout au début de l'algorithme de déchiffrement. Nos simulations d'attaque montrent que cette contremesure arrive à réduire le taux de réussite des attaques par corrélation. En plus, le choix du paramètre r permet une certaine flexibilité : pour plus de robustesse face aux attaques, r peut être augmenté. Par contre, une telle augmentation entraîne un surcoût en terme de surface d'implantation. Selon l'application, une valeur convenable de r peut être choisie en fonction des contraintes matérielles et de la vulnérabilité face aux attaques SCA.

Bibliography

[1] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the 28th annual ACM symposium on Theory of Computing*, pages 99–108, 1996.

[2] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *Proceedings of the 33rd annual ACM Symposium on Theory of Computing*, pages 601–610, 2001.

[3] S. Akleylek, Nina Bindel, Johannes A. Buchmann, Juliane Krämer, and Giorgia Azzurra Marson. An efficient lattice-based signature scheme with provably secure instantiation. In *Proceedings of the 8th International Conference on Cryptology in Africa (AFRICACRYPT)*, pages 44–60, Fes, Morocco, 2016. `https://doi.org/10.1007/978-3-319-31517-1_3`.

[4] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A New Hope. In *Proc. 25th USENIX Security Symposium*, pages 327–343, 2016. `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim`.

[5] Michał Andrzejczak. The Low-Area FPGA Design for the Post-Quantum Cryptography Proposal Round5. In *Proc. Federated Conf. on Computer Science and Information Systems (FedCSIS)*, pages 213–219. IEEE, 2019.

[6] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation, December 2017. `https://hal.archives-ouvertes.fr/hal-01671903`.

[7] AC Atici, Lejla Batina, Benedikt Gierlichs, and Ingrid Verbauwhede. Power analysis on NTRU implementations for RFIDs: First results. *RFIDsec08 : Workshop on RFID Security*, pages 1–11, 2008.

[8] Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and J-P Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 260–275. Springer, 2002.

[9] A. Aysu, Y. Tobah, M. Tiwari, A. Gerstlauer, and M. Orshansky. Horizontal side-channel vulnerabilities of post-quantum key exchange protocols. In *Proc. IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 81–88, May 2018. `https://doi.org/10.1109/HST.2018.8383894`.

[10] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, et al. Supersingular isogeny key encapsulation. *Submission to the NIST Post-Quantum Standardization project*, 2017.

[11] H. Baan, S. Bhattacharya, S. R. Fluhrer, Ó. García-Morchón, T. Laarhoven, R. Rietman, M.J.O. Saarinen, L. Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In *Proceedings of 10th International Conference on Post-Quantum Cryptography (PQCrypto)*, pages 83–102, 2019. `https://doi.org/10.1007/978-3-030-25510-7_5`.

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

[13] Daniel V Bailey, Daniel Coffin, Adam Elbirt, Joseph H Silverman, and Adam D Woodbury. Ntru in constrained devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 262–272. Springer, 2001.

[14] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Advances in Cryptology - EUROCRYPT 2012 - Proceedings of 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 719–737. 2012. `https://doi.org/10.1007/978-3-642-29011-4_42`.

[15] Elaine Barker. Recommendation for key management: Part 1 – general. NIST Special Publication 800-57 Part 1 Revision 5, May 2020. `https://doi.org/10.6028/NIST.SP.800-57pt1r5`.

[16] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. NIST Special Publication 800-56A Revision 3, 2018. `https://doi.org/10.6028/NIST.SP.800-56Ar3`.

[17] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings of Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[18] A.G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):20:1–20:19, January 2012.

[19] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the 27th annual ACM-SIAM symposium on Discrete algorithms*, pages 10–24, 2016.

[20] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic McEliece: conservative code-based cryptography. *Submission to the NIST post quantum standardization process*, 2017.

[21] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: reducing attack surface at low cost. In *Proceedings of International Conference on Selected Areas in Cryptography*, pages 235–260. Springer, 2017.

[22] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems. `https://bench.cr.yp.to`, accessed 16 April 2020.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference. Submission to NIST SHA-3 competition, 2011. `https://keccak.team/files/Keccak-reference-3.0.pdf`.

[24] N. Bindel, J. Buchmann, and J. Kramer. Lattice-based signature schemes and their sensitivity to fault attacks. In *Proceedings of Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, volume 00, pages 63–77, Aug. 2016.

[25] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

[26] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *Proceedings of International Conference on the Theory and Applications of Cryptographic Techniques*, pages 37–51. Springer, 1997.

[27] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, April 2018.

[28] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1006–1018, October 2016. `https://doi.org/10.1145/2976749.2978425`.

[29] Joppe W Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Assessing the feasibility of single trace power analysis of Frodo. In *Proceedings of International Conference on Selected Areas in Cryptography*, pages 216–234. Springer, 2018.

[30] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Proceedings of International workshop on cryptographic hardware and embedded systems (CHES)*, pages 16–29. Springer, 2004.

[31] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, USA, 1992.

[32] 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 *Proc. 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 323–345, August 2016.

[33] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 21–43, 2018.

[34] T. Chabrier and A. Tisserand. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In *Proc. 21st Symposium on Computer Arithmetic (ARITH)*, pages 219–228. IEEE Computer Society, April 2013.

[35] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 13–28. Springer, 2002.

[36] L. Chen, S. Jordan, Yi-Kai Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. Technical report, NIST, 2016. `http://dx.doi.org/10.6028/NIST.IR.8105`.

[37] Yuanmi Chen and Phong Q Nguyen. Bkz 2.0: Better lattice security estimates. In *Proceedings of International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer, 2011.

[38] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *Proceedings of International Conference on Information and Communications Security*, pages 46–61. Springer, 2010.

[39] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[40] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[41] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *International workshop on cryptographic hardware and embedded systems*, pages 292–302. Springer, 1999.

[42] Philippe Coussy and Adam Morawiec. *High-level synthesis: from algorithm to digital circuit.* Springer Science & Business Media, 2008.

[43] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to ideal-SVP. In *Proceedings of International Conference on the Theory and Applications of Cryptographic Techniques*, pages 324–348. Springer, 2017.

[44] Viet B Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign. In *Proc. Int. Conf. on Field-Programmable Technology (ICFPT)*, pages 206–214. IEEE, 2019.

[45] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *Proc. 10th International Conference Progress in Cryptology in Africa (AFRICACRYPT)*, pages 282–305, Marrakesh, Morocco, May 2018.

[46] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Proc. Int. Conf. on Information Security*, pages 171–186. Springer, 2006.

[47] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[48] Libey Djath, Timo Zijlstra, Karim Bigou, and Arnaud Tisserand. Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA. *Conférence d'informatique en Parallélisme, Architecture et Système (Compas)*, 2019.

[49] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In *Proc. 33rd Annual Cryptology Conference CRYPTO*, pages 40–56, Santa Barbara, CA, USA, August 2013. `https://doi.org/10.1007/978-3-642-40041-4_3`.

[50] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018.

[51] Jan-Pieter D'Anvers, Qian Guo, Thomas Johansson, Alexander Nilsson, Frederik Vercauteren, and Ingrid Verbauwhede. Decryption failure attacks on IND-CCA secure lattice-based schemes. In *IACR International Workshop on Public Key Cryptography*, pages 565–598. Springer, 2019.

[52] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on ring/mod-LWE/LWR based schemes. In *International Conference on Post-Quantum Cryptography*, pages 103–115. Springer, 2019.

[53] T. Espitau, P.-A. Fouque, B. Gérard, and M. Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1857–1874, November 2017. `https://doi.org/10.1145/3133956.3134028`.

[54] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. Loop-abort faults on lattice-based signature schemes and key exchange protocols. *IEEE Transactions on Computers*, 67(11):1535–1549, 2018.

[55] Ulrich Fincke and Michael Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44(170):463–471, 1985.

[56] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[57] Nicolas Gama, Phong Q Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 257–278. Springer, 2010.

[58] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography.* Springer Science & Business Media, 2006.

[59] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *Proceedings of International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[60] E. Homsirikamol and K. Gaj. Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study. In *Proc. International Conference on Field Programmable Technology (ICFPT)*, pages 120–127, 2017.

[61] Ekawat Homsirikamol and Kris Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study. In *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–8. IEEE, 2014.

[62] Ekawat Homsirikamol and Kris Gaj. Hardware benchmarking of cryptographic algorithms using high-level synthesis tools: The SHA-3 contest case study. In *Proceedings of International Symposium on Applied Reconfigurable Computing*, pages 217–228. Springer, 2015.

[63] J. Howe, C. Moore, M. O'Neill, F. Regazzoni, T. Güneysu, and K. Beeden. Lattice-based encryption over standard lattices in hardware. In *Proceedings of 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[64] James Howe, Marco Martinoli, Elisabeth Oswald, and Francesco Regazzoni. Optimised lattice-based key encapsulation in hardware. Second NIST Post-Quantum Cryptography Standardization Conference, August 2019.

[65] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):372–393, aug 2018. `https://tches.iacr.org/index.php/TCHES/article/view/7279`.

[66] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. Power analysis on NTRU prime. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 123–151, 2020.

[67] Marc Joye, Arjen K Lenstra, and Jean-Jacques Quisquater. Chinese remaindering based cryptosystems in the presence of faults. *Journal of cryptology*, 12(4):241–245, 1999.

[68] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[69] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete gaussian sampling. *IEEE Transactions on Computers*, 67(11):1561–1571, 2018.

[70] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete gaussian sampling. A case study on the falcon signature scheme. In *Proceedings of the 56th Annual Design Automation Conference*, DAC, New York, USA, 2019. ACM. `https://doi.org/10.1145/3316781.3317887`.

[71] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. 19th Annual International Cryptology Conference (CRYPTO)*, pages 388–397, August 1999.

[72] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[73] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[74] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. Post-quantum key exchange on FPGAs. *IACR Cryptology ePrint Archive*, 2017:690, 2017. `http://eprint.iacr.org/2017/690`.

[75] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes, and Cryptography*, 75(3):565–599, 2015.

[76] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261(ARTICLE):515–534, 1982.

[77] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–5, 2019.

[78] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, J. Großschädl, Howon Kim, and I. Verbauwhede. Efficient ring-LWE encryption on 8-bit AVR processors. In *Proc. 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 663–682, September 2015.

[79] P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Proc. 15th International Conference on Cryptology and Network Security (CANS)*, pages 124–139, November 2016.

[80] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In *Proc. 15th International Workshop on Fast Software Encryption (FSE)*, pages 54–72, February 2008. `https://doi.org/10.1007/978-3-540-71039-4_4`.

[81] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Proc. 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–23. June 2010.

[82] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[83] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[84] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[85] Robert J McEliece. A public-key cryptosystem based on algebraic coding theory. *The Deep Space Network Progress Report*, 42-44:114–116, 1978.

[86] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[87] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based key encapsulation mechanism. *IACR Cryptol. ePrint Arch.*, 2020:321, 2020. `https://www.esat.kuleuven.be/cosic/publications/article-3163.pdf`.

[88] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[89] Michele Mosca. Cybersecurity in an era with quantum computers: will we be ready? *IEEE Security & Privacy*, 16(5):38–41, 2018.

[90] T. Oder and T. Güneysu. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In *Proc. 5th International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, September 2017.

[91] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu. Practical CCA2-secure and masked ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(1):142–174, 2018. `https://doi.org/10.13154/tches.v2018.i1.142-174`.

[92] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *J. ACM*, 15(2):252–264, 1968.

[93] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be: Attacking strongswan's implementation of post-quantum signatures. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1843–1855, 2017.

[94] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *Proceedings of International Conference on Cryptology and Information Security in Latin America*, pages 130–149. Springer, 2019.

[95] John M Pollard. The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[96] T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Proc. 20th International Conference on Selected Areas in Cryptography (SAC)*, pages 68–85, August 2013.

[97] T. Pöppelmann and T. Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *Proc. IEEE International Symposium on Circuits and Systemss (ISCAS)*, pages 2796–2799, June 2014.

[98] T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *Proc. 4th International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 346–365, August 2015.

[99] R. Primas, P. Pessl, and S. Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *Proc. 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 513–533, September 2017.

[100] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proc. 37th Annual ACM Symposium on Theory of Computing*, pages 84–93, May 2005.

[101] O. Reparaz, S. Sinha Roy, R. de Clercq, F. Vercauteren, and I. Verbauwhede. Masking ring-LWE. *J. Cryptographic Engineering*, 6(2):139–153, 2016. `https://www.esat.kuleuven.be/cosic/publications/article-2634.pdf`.

[102] O. Reparaz, S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. A masked ring-LWE implementation. In *Proc. 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 683–702, September 2015. `https://doi.org/10.1007/978-3-662-48324-4_34`.

[103] Oscar Reparaz, Ruan Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-LWE masking. In *Proceedings of the 7th International Workshop on Post-Quantum Cryptography*, PQCrypto, pages 233–244, Berlin, Heidelberg, 2016. Springer-Verlag. `https://www.esat.kuleuven.be/cosic/publications/article-2633.pdf`.

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

[105] S. Sinha Roy, F. Vercauteren, N. Mentens, D. Donglong Chen, and I. Verbauwhede. Compact Ring-LWE cryptoprocessor. In *Proc. 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 371–391, September 2014.

[106] M.-J. O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures - engineering a side-channel resistant post-quantum signature scheme with compact signatures. *J. Cryptographic Engineering*, 8(1):71–84, 2018. `https://doi.org/10.1007/s13389-017-0149-6`.

[107] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1-3):181–199, 1994.

[108] P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.

[109] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, 1999.

[110] C. D. Walter. Montgomery's multiplication technique: How to make it smaller and faster. In *Proc. First International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 80–93, Worcester, MA, USA, August 1999. Springer.

[111] An Wang, Xuexin Zheng, and Zongyue Wang. Power analysis attacks and countermeasures on NTRU-based wireless body area networks. *KSII Transactions on Internet & Information Systems*, 7(5), 2013.

[112] Xilinx. *7 Series DSP48E1 Slice (User Guide UG479, v1.10).* 2018. `https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`.

[113] Xilinx. *High Level Synthesis (User Guide UG902, v2018.3).* 2018. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf`.

[114] Xilinx. *7 Series FPGAs Memory Resources (User Guide UG473, v1.14).* 2019. `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.

[115] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT. *IACR Trans. on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 49–72, 2020.

[116] Yuqing Zhang, Chenghua Wang, Dur E. Shahwar Kundi, Ayesha Khalid, Máire O'Neill, and Weiqiang Liu. An efficient and parallel R-LWE cryptoprocessor. *IEEE Transactions on Circuits and Systems—II: Express Briefs*, 67(5):886–890, 2020.

[117] Timo Zijlstra, Karim Bigou, and Arnaud Tisserand. FPGA Implementation and Comparison of Protections against SCAs for RLWE. In *Proc. Int. Conf. on Cryptology in India (IndoCrypt)*, December 2019. `https://hal.archives-ouvertes.fr/hal-02309481/file/camera_ready.pdf`.

UNIVERSITE
BRETAGNE
LOIRE / MATHSTIC

Université Bretagne Sud

Titre : Accélérateurs matériels sécurisés pour la cryptographie post-quantique.

Mot clés : cryptographie à base de réseaux euclidiens, protections contre des attaques par canaux auxiliaires, implantation matérielle HLS sur FPGA, LWE, RLWE, MLWE

Résumé : L'algorithme quantique de Shor peut être utilisé pour résoudre le problème de factorisation de grands entiers et le logarithme discret dans certains groupes. La sécurité des protocoles cryptographiques à clé publique les plus répandus dépend de l'hypothèse que ces problèmes sont difficiles à résoudre. La cryptographie post-quantique est basée sur des problèmes mathématiques difficiles à résoudre même pour des ordinateurs quantiques, tels que *Learning with Errors* (LWE) et ses variantes RLWE et MLWE. Dans cette thèse, nous présentons et comparons des implantations en HLS sur FPGA d'algorithmes de chiffrement à clé publique basés sur LWE, RLWE et RLWE. Nous discutons des compromis entre sécurité, temps de calcul et coût en surface. Les implantations sont parallélisées afin d'obtenir une accélération importante. Nous analysons la sécurité matérielle de nos implantations, et proposons des protections contre des attaques par canaux auxiliares. Nous améliorons des contre-mesures de l'état de l'art, telles que le masquage, et nous proposons également de nouvelles protections basées sur la représentation redondante des nombres et sur des permutations aléatoires des opérations de calcul. Toutes ces protections sont implantées et évaluées sur FPGA dans le but de comparer leurs coûts et performances.

Title: Secure Hardware Accelerators for Post Quantum Cryptography.

Keywords: lattice based cryptography, protection against side-channel attacks, HLS hardware implementation on FPGA, LWE, RLWE, MLWE

Abstract: Shor's quantum algorithm can be used to efficiently solve the integer factorisation problem and the discrete logarithm in certain groups. The security of the most commonly used public key cryptographic protocols relies on the conjectured hardness of exactly these mathematical problems. *Post quantum cryptography* relies on mathematical problems that are computationally hard for quantum computers, such as Learning with Errors (LWE) and its variants RLWE and MLWE. In this thesis, we present and compare FPGA implementations using HLS of LWE, RLWE and MLWE based public-key encryption algorithms. We discuss various trade-offs between security, computation time and hardware cost. The implementations are parallelized in order to obtain maximal speed-up. We also discuss hardware security and propose countermeasures against side channel attacks. We consider countermeasures from the state of the art, such as masking, and propose improvements to these algorithms. Moreover, we propose new countermeasures based on redundant number representation and random shuffling of operations. All our countermeasures are implemented and evaluated on FPGA to compare their cost and performance.